Distribution
Category UC-411

SAND98-8206 (revised)

Unlimited Release

First Printed November 1997

Jess, The Expert System Shell for the Java Platform

http://herzberg.ca.sandia.gov/jess

Ernest J. Friedman-Hill

Distributed Computing Systems

Sandia National Laboratories

Livermore, CA

Version 6.1RC1 (24 March 2003)

ABSTRACT

This report describes Jess, an expert system shell and scripting language written entirely in Sun Microsystem's Java language. Jess supports the development of rule-based expert systems which can be tightly coupled to code written in the powerful, portable Java language. The syntax of the Jess language is discussed, and a comprehensive list of supported functions is presented. Guides to calling Java functions from Jess, to extending Jess by writing Java code, and to embedding Jess in Java applications are also included.

Abbreviated Table of Contents

Embedding Jess in Java Code
: A brief discussion of Java application structure

Writing GUIs in Jess
: How to create GUIs without writing a single line of Java code

The Jess Function List
: Reference documentation for every function in the Jess language

Java API Guide
: Javadoc-generated documentation for the Java classes that comprise Jess

The Rete Algorithm
: An introduction to the technical details of how Jess works

Change History
: How Jess got to where it is today

Useful Reference Material
: A bibliography of books, articles and links useful to the Jess user

Release Notes
: Information about bugs and limitations specific to this version of Jess, and information about porting applications from Jess 5 to Jess 6

Function Cross-reference
: A list of all Jess functions organized by usage

Table of Contents

Back to index

1. Introduction

1.1. Abstract

This report describes Jess, an expert system shell and scripting language written entirely in Sun Microsystem's Java language. Jess supports the development of rule-based expert systems which can be tightly coupled to code written in the powerful, portable Java language. The syntax of the Jess language is discussed, and a comprehensive list of supported functions is presented. Guides to calling Java functions from Jess, to extending Jess by writing Java code, and to embedding Jess in Java applications are also included.

1.2. Compatibility

Jess 6.1 is compatible with all versions of Java starting with Java 1.2. In particular, this includes JDK 1.4 (or "Java 2" as it is now known.) Versions numbered 4.x are compatible with JDK 1.0, and the 5.x versions work with JDK 1.1.

> When compiling Jess with JDK 1.4, you will see warnings concerning the use of the new keyword `assert` as a method name. This is normal -- note that these are warnings, not errors. The `assert` method is deprecated in Jess 6.1 and will be removed in Jess 7.0. Until then, these warnings are harmless.

1.3. Mailing List

There is a Jess email discussion list you can join. To get information about the jess-users list, send a message to majordomo@sandia.gov containing the text

```
help
info jess-users
end
```

as the body of the message. There is an archive of the list at
http://www.mail-archive.com/jess-users@sandia.gov .

1.4. Bugs

This is the first Jess 6.1 release candidate. There may still be bugs. Please read the release notes for specific information. Comments and bug reports are welcome. Contact me at `ejfried@ca.sandia.gov` so I can fix them for a later release.

1.5. Assumptions

Jess is a programmer's library. The library itself is written in Java. The library serves as an interpreter for another language, which I will refer to in this document as the Jess language. The Jess language is very similar to the language defined by the CLIPS expert system shell, which in turn is a highly specialized form of LISP.

Therefore, I am going to assume that you, the reader, are a programmer who will be using either one or both of these languages. I will assume that all readers have at least a minimal facility with Java. You must have a Java compiler and runtime system, and you must know how to use it at least in a simple way. You should know how to use it to

- compile a collection of Java source files
- run a Java application
- deal with configuration issues like the CLASSPATH variable

If you do not have at least this passing familiarity with a Java environment, then may I suggest you purchase an introductory book on the topic.

For those readers who are going to program in the Jess language, I assume general familiarity with the principles of programming. I will describe the entire Jess language, so no familiarity with LISP is required. Furthermore, I will attempt to describe, to the extent possible, the important concepts of rule-based systems as they apply to Jess. Again, though, I will assume that the reader has some familiarity with these concepts and more. If you are unfamiliar with rule-based systems, you may want to purchase a text on this topic as well.

Many readers will want to extend Jess' capabilities by either adding commands (written in Java) to the Jess language, or embedding the Jess library in a Java application. Others will want to use the Jess language's Java integration capabilities to call Java functions from Jess language programs. In sections of this document targeted towards these readers, I will assume moderate knowledge of Java programming. I will not teach any aspects of the Java language. The interested reader is once again referred to your local bookstore.

This document contains a bibliography wherein a number of books on all these topics are listed.

1.6. Getting ready

1.6.1. Unpacking the Distribution

If you download Jess for UNIX, you can extract the files using tar and gunzip:

```
gunzip Jess61RC1.tgz
tar xf Jess61RC1.tar
```

If you downloaded Jess for Windows, you get a .zip file which should be unzipped using a Win32-aware unzip program like WinZip. Don't use PKUNZIP since it cannot handle long file names.

When Jess is unpacked, you should have a directory named `Jess61RC1/`. There are two kinds of Jess distributions: *binary-only* and *source.* Inside this directory should be the following files and subdirectories, depending on which type of distribution you have:

`docs/`	This documentation
`jess/`	A directory containing the `jess` package. There are many source files in here that implement Jess's inference engine. Others implement a number of Jess GUIs and command-line interfaces. `Main.java` implements the Jess command-line interface. `Console.java` is a very simple GUI console for Jess; `ConsoleApplet.java` is an applet version of the same. If you have a binary-only distribution of Jess, this directory will contain only the `examples` subdirectory.
`examples/`	A directory of tiny example Jess files.
`jess/examples`	A directory of more complicated examples, containing example Java source files.
`jess.jar (optional)`	A Java archive file containing the Jess classes themselves. Binary distribution only.
`Makefile (optional)`	A simple makefile for Jess. Source distribution only.

1.6.2. Compiling Jess

If you have a source distribution of Jess, you have a set of Java source files, and you'll need to compile them first before you can run Jess. If you have a binary distribution, you can skip this section. If you have a `make` utility (any UNIX-like `make`; you could use the CygWin environment on Windows), you can just run `make` and the enclosed Makefile will build everything. You will have to edit it a bit first to specify the path to your Java compiler. Otherwise, you can compile Jess by typing a few commands yourself. Using Sun's JDK on some version of Windows, the commands

```
javac -d . jess\*.java
javac -d . jess\awt\*.java
javac -d . jess\factory\*.java
```

would work just fine, given that `Jess61RC1/` is your current directory.

> *NOTE: Jess works fine with JDK 1.4, but you will get warnings during the compilation about a conflict between the new Java keyword "assert" amd the Jess function `jess.Rete.assert()`. These are just warnings, and they don't stop the compilation. This function is now deprecated in Jess and will be phased out over the next few versions.*

If you have problems, be sure that if you have the `CLASSPATH` environment variable set, it includes '.', the current directory. *Don't* try to compile from inside the `Jess61RC1/jess/` directory; it won't work.

If you're on a UNIX system instead of a Windows system, you can use the commands given above, but you'll need to change the backslashes (\) into forward slashes (/).

You must use a Java 2 compiler to compile Jess. The resulting code will run on any Java 2 or later VM. Jess works great with JDK 1.3.

There are a number of optional example source files in the subdirectories `Jess61RC1/jess/examples/` that aren't compiled if you follow the instructions above. You can compile the examples one at a time. For example, to compile the example named `pumps` using the JDK on a Windows system, you can use the command

```
javac -d . jess\examples\pumps\*.java
```

Again, *don't* set your current directory to, for example, Jess61RC1/jess/examples/pumps/ to compile the pumps example: it will *not* work. The compiler will report all sorts of errors about classes not being found and the jess package not being found. Compile everything from the Jess61RC1 directory. I can't stress this enough: this is by far the most common problem people have in getting started with Jess!

I personally use the Jikes Java compiler from IBM. The compiler itself is very fast -- it compiles all of Jess in just a few seconds on my machine. I highly recommend it, and it's free!

1.6.3. Jess Example Programs

There are a few trivial example programs (in the examples/ directory) that you can use to confirm that you have properly compiled Jess. These include `fullmab.clp`, `zebra.clp`, and `wordgame.clp`. `fullmab.clp` is a version of the classic Monkey and Bananas problem. To run it yourself from the command line, just type:

```
java jess.Main examples/fullmab.clp
```

(if you've got a source distribution) or

```
java -classpath jess.jar jess.Main examples/fullmab.clp
```

(if you've got a binary-only distribution) and the problem should run, producing a few screens of output. Any file of Jess code can be run this way. Many simple CLIPS programs will also run unchanged in Jess. Note that giving Jess a file name on the command line is like using the `batch` command in CLIPS. Therefore, you generally need to make sure that the file ends with:

```
(reset)
(run)
```

or no rules will fire. The `zebra.clp` and `wordgame.clp` programs are two classic CLIPS examples selected to show how Jess deals with tough situations. These examples both generate large numbers of partial pattern matches, so they are slow and use up a lot of memory. Other examples include `sticks.clp` (an interactive game), `frame.clp` (a demo of building a graphical interface using Jess's Java integration capabilities), and `animal.clp`. Note that `animal.clp` is hardwired to expect a data file to exist in a subdirectory examples/ of the current directory.

In the `jess/examples/*` subdirectories, you will find some more complex examples, all of which contain both Java and Jess code. As such, these are generally examples of how to tie Jess and Java together. The *Pumps* examples is a full working program that demonstrates how Jess rules can react to the properties of Java Beans.

1.6.4. Command-line Interface

Jess has an interactive command-line interface. Just type `java jess.Main` (or `java -classpath jess.jar jess.Main`) to get a `Jess>` prompt. To execute a file of CLIPS code from the command prompt, use the `batch` command:

```
Jess> (batch examples/sticks.clp)
Who moves first (Computer: c Human: h)?
```

Note that in the preceding example, you type what follows the `Jess>` prompt, and Jess responds with the text on the next line. I will follow this convention throughout this manual.

You can use the Jess `system` command to invoke an editor from the Jess command line to edit a file of Jess code before reading it in with `batch`. `system` also helps to allow non-Java programmers to integrate Jess with other applications. Given that you have an application named `xlogo` on your system, try:

```
Jess> (system xlogo &)
  <External-Address:java.lang.UNIXProcess>
```

The & character makes the program run in the background. Omitting it will keep the system command from returning until the called program exits. The `system` command returns the Java Process object representing the launched application.

The class `jess.Console` is a graphical version of the Jess command-line interface. You type into a text field at the bottom of the window, and output appears in a scrolling window above. Type `java jess.Console` to try it.

1.6.5. Jess as an Applet

The class `jess.ConsoleApplet` is a generic Jess applet that uses the same display as the `jess.Console` class. It can be used in general question-and-answer situations simply by embedding the applet class on a Web page. The applet accepts two applet parameters. The value of an `INPUT` parameter will be interpreted as a Jess program to run when starting up. Note that when this program halts, the Jess prompt will appear in the applet window. The applet also accept a `COMPACT` parameter. If present, `ConsoleApplet` will contain only a bare-bones version of Jess (no optional functions will be loaded).

Since Jess 6 uses the Java 2 API, it won't work in the native JVM of most deployed web browsers. Netscape 4.x and all versions of Microsoft Internet Explorer use some version of a JDK 1.1 Java Virtual Machine. You can use Jess 4 or 5 in these browsers, or you can require the user to download the Java Plug-In. A full discussion of this topic is beyond the scope of this document -- you're encouraged to get a book that covers deploying applets on the Web if you're interested.

Note that even in Jess 4 and 5, the ConsoleApplet and ConsoleDisplay classes use the Java 1.1 event model, which is still not supported by some of the installed base of Web browsers; the Plug-in might still be necessary. Don't use `ConsoleApplet` if you want to deploy highly portable applets! Actually, the idea of deploying Jess as an applet makes less and less sense these days; a much better alternative is to run Jess on the server side (as a servlet, for example) and run only the GUI on the client. Good applets are generally very small (a few tens of kilobytes), while Jess's class files now occupy hundreds of kilobytes.

1.7. What makes a good Jess application?

Jess can be used in two overlapping ways. First, it can be a rule engine - a special kind of program that very efficiently applies rules to data. A rule-based program can have hundreds or even thousands of *rules*, and Jess will continually apply them to data in the form of a *knowledge base*. Often the rules will represent the heuristic knowledge of a human expert in some domain, and the knowledge base will represent the state of an evolving situation (an interview, an emergency). In this case, they are said to constitute an *expert system.* Expert systems are widely used in many domains. Among the newest applications of expert systems are as the reasoning part of *intelligent agents*, in enterprise resource planning (ERP) systems, and in order validation for electronic commerce.

But the Jess language is also a general-purpose programing language, and furthermore, it can directly access all Java classes and libraries. For this reason, Jess is also frequently used as a dynamic scripting or rapid application development environment. While Java code generally must be compiled before it can be run, a line of Jess code is executed immediately upon being typed. This allows you to experiment with Java APIs interactively, and build up large programs incrementally. It is also very easy to extend the Jess language with new commands written in Java or in Jess itself, and so the Jess language can be customized for specific applications.

Jess is therefore useful in a wide range of situations. One application for which Jess is not so well suited is as an applet intended for Internet use. Jess's size (a few hundred kilobytes of compiled code) makes it too large for applet use except on high-speed LANs. Furthermore, some of Jess's capabilities are lost when it is used in a browser: for example, access to Java APIs from the Jess language may not work at all due to security restrictions in some browsers. When building Web-based applications using Jess, you should strongly consider using Jess on the server side (in a servlet, for example.)

1.7.1. Jess vs. Prolog

As in all pursuits, in programming you should choose the right tool for the right job. Prolog and a Rete-based system like Jess are very different. The central concept in Prolog is backwards chaining: given the rules

```
human(Socrates).
mortal(X) :- human(X).
```

you might be interested in knowing if `mortal(Socrates)` was true. Prolog uses the rules to find it by looking for `human(Socrates).` Note that if you forget the result and ask for it again, Prolog has to compute it again.

The central concept in Jess, though, is forwards chaining. Here, you have

```
Jess> (assert (human Socrates))
  Jess> (defrule mortal (human ?X) => (assert (mortal ?X)))
  Jess> (watch facts)
  Jess> (run)
   ==> f-1 (MAIN::mortal Socrates)
  1
```

You don't specifically want to know `(mortal Socrates)` but rather you want to know what happens given that `(human Socrates)` is known. `(mortal Socrates)` is a result. After the rule has fired,

`(mortal Socrates)` is known, and the rule `mortal` never has to assert this fact again.

One more difference is that Prolog is really meant to be used from the console; i.e., you're actually supposed to sit down and type `mortal(Socrates).` In Jess, only developers do this; the command line is not intended for end-users. Prolog is really about answering queries, while Jess is about acting in response to inputs.

Jess is different than some Rete-based systems in that it includes both a kind of backwards chaining and a construct called defquery which lets you make direct queries of the knowledge base. Both of these help Jess a better fit for some Prolog applications, but they don't make Jess into a Prolog-like system. Prolog is optimized, in a sense, for space, at the cost of speed. Jess (and its Rete algorithm) is optimized for speed at the cost of space. The Rete algorithm is all about computing things -once- so they never need to be recomputed, and then reusing them. Prolog's approach is targeted at exploring large numbers of possibilities once, while Rete is aimed at exploring medium-sized numbers of possibilities repeatedly.

Regarding different ways to express the kinds of relationships Prolog can express: Jess offers a rich set of possiblities. Here's one in which the mortality is encoded directly into the facts, so it never needs to be computed at all:

```
Jess> (deftemplate being (slot name))
  Jess> (deftemplate mortal extends being)
  Jess> (deftemplate immortal extends being)
  Jess> (deftemplate monster extends mortal)
  Jess> (deftemplate human extends mortal)
  Jess> (deftemplate god extends immortal)

  Jess> (defrule list-all-humanoids
    ;; fire for all beings, gods, monsters, and humans
    (being (name ?n))
    =>
    (printout t ?n " is a being " crlf))

  Jess> (defrule list-all-mortals
    ;; fires only for mortal things
    (mortal (name ?n))
    =>
    (printout t ?n " is mortal " crlf))

  Jess> (deffacts beings (human (name Bob)) (monster (name Gollum))
                         (god (name Zeus)))

  Jess> (reset)
  Jess> (run)
    Zeus is a being
  Gollum is mortal
  Gollum is a being
  Bob is a being
  Bob is mortal
  5
```

Here's another that's closer in spirit to the Prolog example.

```
Jess> (deftemplate thing (slot type) (slot name))

  Jess> (deffacts things
    (thing (type human) (name Socrates))
    (thing (type mineral) (name Slate))
    (thing (type vegetable) (name Carrot))
    (thing (type dog) (name Rover))
    (thing (type human) (name Bob)))

  Jess> (deffacts mortality
    (mortal human)
    (mortal dog))

  Jess> (defrule list-all-mortals
    ;; fires for dogs and humans
    (mortal ?type)
    (thing (type ?type) (name ?n))
    =>
    (printout t ?n " is mortal." crlf))
  Jess> (reset)
  Jess> (run)
    Rover is mortal.
  Bob is mortal.
  Socrates is mortal.
  3
```

There's a fact that expresses that humans are mortal, and one for each human known. In this example, no extra facts are generated. Nevertheless, the mortality of Socrates is remembered (in the Rete network) and may be used to optimize some later computation.

1.8. About Jess and performance

Jess's rule engine uses an improved form of a well-known algorithm called Rete (latin for "net") to match rules against the knowledge base. Jess is actually faster than some popular expert system shells written in C, especially on large problems, where performance is dominated by algorithm quality.

Note that Rete is an algorithm that explicitly trades space for speed, so Jess' memory usage is not inconsiderable. Jess does contain some commands which will allow you to sacrifice some performance to decrease memory usage. Nevertheless, Jess' memory usage is not ridiculous, and moderate-sized programs will fit easily into Java's default 16M heap.

1.8.1. Sun's HotSpot Virtual Machine

Because Jess is a memory-intensive application, its performance is sensitive to the behavior of the Java garbage collector. Recent JVMs from Sun feature an advanced Java runtime called HotSpot which includes a flexible, configurable garbage collection subsystem. Excellent articles on GC performance tuning are available at Sun's web site. Although every Jess rule base is different, in general, Jess will benefit if the heap size and the object nursery size are each set larger than the default. For example, on my machine, Jess' performance on the Miranker *manners* benchmark with 90 guests is improved by 25% by increasing the initial heap size and nursery size to 32 and 16 megabytes, respectively, from their defaults of 16 meg and 640K. You can do this using

```
java -XX:NewSize=16m -Xms32m -Xmx32m jess.Main <scriptfile>
```

Note that the object nursery is a subset of the Java heap set aside for recently-allocated objects; the total heap size in this example is 32M, not 48M.

1.9. Command-line, GUI, or embedded?

As we've discussed, Jess can be used in many ways. Besides the different categories of problems Jess can be applied to, being a library, it is amenable to being used in many different kinds of Java programs. Jess can be used in command-line applications, GUI applications, servlets, and applets. Furthermore, Jess can either provide the Java `main()` for your program, or you can write it yourself. You can develop Jess applications (with or without GUIs) without compiling a single line of Java code. You can also write Jess applications which are controlled entirely by Java code you write, with a minumum of Jess language code.

The most important step in developing a Jess application is to choose an architecture from among the almost limitless range of possibilities. One way to organize the possibilities is to list them in increasing order of the amount of Java programming involved.

1. Pure Jess language scripts. No Java code at all.
2. Pure Jess language scripts, but the scripts access Java APIs.
3. Mostly Jess language scripts, but some custom Java code in the form of new Jess commands written in Java.
4. Half Jess language scripts, with a substantial amount of Java code providing custom commands and APIs; `main()` provided by Jess.
5. Half Jess language scripts, with a substantial amount of Java code providing custom commands and APIs; `main()` written by you.
6. Mostly Java code, which loads Jess language scripts at runtime.
7. All Java code, which maniulates Jess entirely through its Java API. This option is not fully supported at this time, but will in a future release.

Examples of some of these types of applications are package with Jess. The basic examples like `wordgame.clp`, `zebra.clp`, and `fullmab.clp` are all type 1) programs. `draw.clp` and `frame.clp` are type 2) programs. The `pumps` example is packaged two ways. If you run it using the script file `pumps.clp`, it is a type 4) program; if you run it using `MainInJava.java`, it is a type 6) application.

Your choice can be guided by many factors, but ultimately it will depend on what you feel most comfortable with. Types 4) and 5) are most prevalent in real-world applications.

Back to index

2. The Jess Language

I'm using an extremely informal notation to describe syntax. Basically strings in <angle-brackets> are some kind of data that must be supplied; things in [square brackets] are optional, things ending with `+` can appear one or more times, and things ending with `*` can appear zero or more times.

In general, input to Jess is free-format. Newlines are generally not significant and are treated as whitespace; exceptions will be noted.

2.1. Basics

2.1.1. Atoms

The atom or symbol is a core concept of the Jess language. Atoms are very much like identifiers in other languages. A Jess atom can contain letters, numbers, and the following punctuation: `$*=+/<>_?#.` . An atom may not begin with a number; it may begin with some punctuation marks (some have special meanings as operators when they appear at the start of an atom).

Jess atoms are case sensitive: `foo`, `FOO` and `Foo` are all different atoms.

The best atoms consist of letters, numbers, underscores, and dashes; dashes are traditional word separators. The following are all valid atoms:

```
foo first-value contestant#1 _abc
```

There are three "magic" atoms that Jess interprets specially: `nil`, which is somewhat akin to Java's `null` value; and `TRUE` and `FALSE`, which are Jess' boolean values.

2.1.2. Numbers

Jess uses the Java functions `java.lang.Integer.parseInt` and `java.lang.Double.parseDouble` to parse integer and floating point numbers, respectively. See the documentation for those methods for a precise syntax description. The following are all valid numbers:

```
3 4. 5.643 6.0E4 1D
```

2.1.3. Strings

Character strings in Jess are denoted using double quotes (`"`). Backslashes (`\`) can be used to escape embedded quote symbols. Note that Jess strings are unlike Java strings in several important ways. First, no "escape sequences" are recognized. You cannot embed a newline in a string using "`\n`", for example. On the other hand, real newlines are allowed inside double-quoted strings; they become part of the string. The following are all valid strings:

```
"foo" "Hello, World" "\"Nonsense,\" he said firmly." "Hello,
There"
```

The last string is equivalent to the Java string "Hello,\nThere".

2.1.4. Lists

Another fundamental unit of syntax in Jess is the list. A list always consists of an enclosing set of parentheses and zero or more atoms, numbers, strings, or other lists. The following are valid lists:

```
(+ 3 2) (a b c) ("Hello, World") () (deftemplate foo (slot bar))
```

The first element of a list (the *car* of the list in LISP parlance) is often called the list's *head* in Jess.

2.1.5. Comments

Programmer's comments in Jess begin with a semicolon (`;`) and extend to the end of the line of text. Here is an example of a comment:

```
; This is a list
(a b c)
```

Comments can appear anywhere in a Jess program.

2.2. Functions

As in LISP, all code in Jess (control structures, assignments, procedure calls) takes the form of a function call.

Function calls in Jess are simply lists. Function calls use a prefix notation; a list whose head is an atom that is the name of an existing function can be a function call. For example, an expression that uses the `+` function to add the numbers `2` and `3` would be written `(+ 2 3)`. When evaluated, the value of this expression is the number `5` (not a list containing the single element `5`!). In general, expressions are recognized as such and evaluated in context when appropriate. You can type expressions at the `Jess>` prompt. Jess evaluates the expression and prints the result:

```
Jess> (+ 2 3)
  5
  Jess> (+ (+ 2 3) (* 3 3))
  14
```

Note that you can nest function calls; the outer function is responsible for evaluating the inner function calls.

Jess comes with a large number of built-in functions that do everything from math, program control and string manipulations, to giving you access to Java APIs.

One of the most commonly used functions is `printout`. `printout` is used to send text to Jess's standard output, or to a file. A complete explanation will have to wait, but for now, all you need to know is contained in the following example:

```
Jess> (printout t "The answer is " 42 "!" crlf)
  The answer is 42!
```

Another useful function is `batch`. `batch` evaluates a file of Jess code. To run the Jess source file

`examples/hello.clp` you can enter

```
Jess> (batch examples/hello.clp)
  Hello, world!
```

Each of these functions (along with all the others) is described more thoroughly in the Jess function guide.

2.3. Variables

Programming variables in Jess are atoms that begin with the question mark (`?`) character. The question mark is part of the variable's name. A normal variable can refer to a single atom, number, or string. A variable whose first character is instead a `$` (for example, `$?X`) is a *multivariable*, which can refer to a special kind of list called a *multifield*. You assign to any variable using the `bind` function:

```
Jess> (bind ?x "The value")
  "The value"
```

Multifields are generally created using special multifield functions like `create$` and can then be bound to multivariables:

```
Jess> (bind $?grocery-list (create$ eggs bread milk))
  (eggs bread milk)
```

Variables need not (and cannot) be declared before their first use (except for special variables called defglobals).

Note that to see the value of a variable at the `Jess>` prompt, you can simply type the variable's name.

```
Jess> (bind ?a 123)
  123
  Jess> ?a
  123
```

2.3.1. Global variables (or defglobals)

Any variables you create at the Jess> prompt, or at the "top level" of any Jess language program, are cleared whenever the `reset` command is issued. This makes them somewhat transient; they are fine for scratch variables but are not persistent global variables in the normal sense of the word. To create global variables that are not destroyed by `reset`, you can use the `defglobal` construct.

```
  (defglobal [?<global-name> = <value>]+)
```

Global variable names must begin and end with an asterisk. Valid global variable names look like

```
  ?*a*    ?*all-values*    ?*counter*
```

When a global variable is created, it is initialized to the given value. When the `reset` command is subsequently issued, the variable *may* be reset to this same value, depending on the current setting of the `reset-globals` property. There is a function named `set-reset-globals` that you can use to set

this property. An example will help.

```
Jess> (defglobal ?*x* = 3)
  TRUE
  Jess> ?*x*
  3
  Jess> (bind ?*x* 4)
  4
  Jess> ?*x*
  4
  Jess> (reset)
  TRUE
  Jess> ?*x*
  3
  Jess> (bind ?*x* 4)
  4
  Jess> (set-reset-globals nil)
  FALSE
  Jess> (reset)
  TRUE
  Jess> ?*x*
  4
```

You can read about the `set-reset-globals` and the accompanying `get-reset-globals` function in the Jess function guide.

2.4. Deffunctions

You can define your own functions using the `deffunction` construct. A `deffunction` construct looks like this:

```
  (deffunction <function-name> [<doc-comment>] (<parameter>*)
     <expr>*
    [<return-specifier>])
```

The `<function-name>` must be an atom. Each `<parameter>` must be a variable name. The optional `<doc-comment>` is a double-quoted string that can describe the purpose of the function. There may be an arbitrary number of `<expr>` expressions. The optional `<return-specifier>` gives the return value of the function. It can either be an explicit use of the `return` function or it can be any value or expression. Control flow in `deffunction`s is achieved via control-flow functions like `foreach`, `if`, and `while`. The following is a `deffunction` that returns the larger of its two numeric arguments:

```
Jess> (deffunction max (?a ?b)
    (if (> ?a ?b) then
        (return ?a)
    else
        (return ?b)))
  TRUE
```

Note that this could have also been written as:

```
Jess> (deffunction max (?a ?b)
    (if (> ?a ?b) then
        ?a
     else
        ?b))
  TRUE
```

This function can now be called anywhere a Jess function call can be used. For example

```
Jess> (printout t "The greater of 3 and 5 is " (max 3 5) "." crlf)
  The greater of 3 and 5 is 5.
```

Normally a `deffunction` takes a specific number of arguments. To write a `deffunction` that takes an arbitrary number of arguments, make the last formal parameter be a multifield variable. When the `deffunction` is called, this multifield will contain all the remaining arguments passed to the function. A `deffunction` can accept no more than one such wildcard argument, and it must be the last argument to the function.

2.5. Defadvice

Sometimes a Jess function won't behave exactly as you'd like. The `defadvice` construct lets you write some Jess code which will be executed before or after each time a given Jess function is called. `defadvice` lets you easily "wrap" extra code around any Jess function, such that it executes before (and thus can alter the argument list seen by the real function, or short-circuit it completely by returning a value of its own) or after the real function (and thus can see the return value of the real function and possibly alter it.) defadvice provides a great way for Jess add-on authors to extend Jess without needing to change any internal code.

Here are some examples of what defadvice looks like.

This intercepts calls to 'plus' (+) and adds the extra argument '1', such that (+ 2 2) becomes (+ 2 2 1) -> 5. The variable '$?argv' is special. It always refers to the list of arguments the real Jess function will receive when it is called.

```
Jess> (defadvice before + (bind $?argv (create$ $?argv 1)))
  TRUE
  Jess> (+ 2 2)
  5
```

This makes all additions equal to 1. By returning, the defadvice keeps the real function from ever being called.

```
Jess> (defadvice before + (return 1))
  TRUE
  Jess> (+ 2 2)
  1
```

This subtracts one from the return value of the + function. ?retval is another magic variable - it's the value the real function returned. When we're done, we remove the advice with `undefadvice`.

```
Jess> (defadvice after + (return (- ?retval 1)))
  TRUE
  Jess> (+ 2 2)
  3
  Jess> (undefadvice +)
  Jess> (+ 2 2)
  4
```

2.6. Java reflection

Among the list of functions above are a set that let you create and manipulate Java objects directly from Jess. Using them, you can do virtually anything you can do from Java code, except for defining new classes. Here is an example in which I create a Java `Hashtable` and add a few String objects to it, then lookup one object and display it.

```
Jess> (bind ?ht (new java.util.Hashtable))
  <External-Address:java.util.Hashtable>
  Jess> (call ?ht put "key1" "element1")
  Jess> (call ?ht put "key2" "element2")
  Jess> (call ?ht get "key1")
  "element1"
```

As you can see, Jess converts freely between Java and Jess types when it can. Java objects that can't be represented as a Jess type are called *external address values*. The `Hashtable` in the example above is one of these.

Jess can also access member variables of Java objects using the `set-member` and `get-member` functions.

```
Jess> (bind ?pt (new java.awt.Point))
  <External-Address:java.awt.Point>
  Jess> (set-member ?pt x 37)
  37
  Jess> (set-member ?pt y 42)
  42
  Jess> (get-member ?pt x)
  37
```

You can access static members by using the name of the class instead of an object as the first argument to these functions.

```
Jess> (get-member System out)
  <External-Address:java.io.PrintStream>
```

Note that we don't have to say "java.lang.System." The java.lang package is implicitly "imported" much as it is in Java code. Jess also has an `import` function that you can use explicitly.

Jess converts values from Java to Jess types according to the following table.

Java type	Jess type
A null reference	The atom ’nil’
A void return value	The atom ’nil’
String	RU.STRING
An array	A Jess multifield
boolean or java.lang.Boolean	The atoms ’TRUE’ and ’FALSE’
byte, short, int, or their wrappers	RU.INTEGER
long or Long	RU.LONG
double, float or their wrappers	RU.FLOAT
char or java.lang.Character	RU.ATOM
anything else	RU.EXTERNAL_ADDRESS

Jess converts values from Jess to Java types with some flexibility, according to this table. Generally when converting in this direction, Jess has some idea of a *target type;* i.e., Jess has a `java.lang.Class` object and a `jess.Value` object, and wants to turn the `Value`’s contents into something assignable to the type named by the `Class`. Hence the atom ’TRUE’ could be passed to a function expecting a boolean argument, or to one expecting a String argument, and the call would succeed in both cases.

Jess type	Possible Java types
RU.EXTERNAL_ADDRESS	The wrapped object
The atom ’nil’	A null reference
The atoms ’TRUE’ or ’FALSE’	java.lang.Boolean or boolean
RU.ATOM, RU.STRING	String, char, java.lang.Character
RU.FLOAT	float, double, and their wrappers
RU.INTEGER	long, short, int, byte, char, and their wrappers
RU.LONG	long, short, int, byte, char, and their wrappers
RU.LIST	A Java array

Sometimes you might have trouble calling overloaded methods - for example, passing the String "TRUE" to a Java method that is overloaded to take either a boolean or a String. In this case, you can always resort to using an explicit wrapper class - in this case, passing a `java.lang.Boolean` object should fix the problem.

To learn more about the syntax of `call`, `new`, `set-member`, `get-member`, and other Java integration functions, see the Jess function guide.

2.7. The knowledge base

A rule-based system maintains a collection of knowledge nuggets called *facts.* This collection is known as the *knowledge base.* It is somewhat akin to a relational database, especially in that the facts must have a specific structure. In Jess, there are three kinds of facts: *ordered facts*, *unordered facts*, and *definstance facts.*

2.7.1. Ordered facts

Ordered facts are simply lists, where the first field (the *head* of the list) acts as a sort of category for the fact. Here are some examples of ordered facts:

```
(shopping-list eggs milk bread)
(person "Bob Smith" Male 35)
(father-of danielle ejfried)
```

You can add ordered facts to the knowledge base using the `assert` function. You can see a list of all the facts in the knowledge base using the `facts` command. You can completely clear Jess of all facts and other data using the `clear` command.

```
Jess> (reset)
  TRUE
  Jess> (assert (father-of danielle ejfried))
  <Fact-1>
  Jess> (facts)
  f-0   (MAIN::initial-fact)
  f-1   (MAIN::father-of danielle ejfried)
  For a total of 2 facts.
```

As you can see, each fact is assigned an integer index (the *fact-id*) when it is asserted. You can remove an individual fact from the knowledge base using the `retract` function.

```
Jess> (retract (fact-id 1))
  TRUE
  Jess> (facts)
  f-0   (MAIN::initial-fact)
  For a total of 1 facts.
```

The fact `(initial-fact)` is asserted by the `reset` command. It is used internally by Jess to keep track of its own operations; you should generally not retract it.

2.7.2. Unordered facts

Ordered facts are useful, but they are unstructured. Sometimes (most of the time) you need a bit more organization. In object-oriented languages, *objects* have named *fields* in which data appears. Unordered facts offer this capability (although the fields are traditionally called *slots.*)

```
   (person (name "Bob Smith") (age 34) (gender Male))
   (automobile (make Ford) (model Explorer) (year 1999))
```

before you can create unordered facts, you have to define the slots they have using the `deftemplate` construct:

```
  (deftemplate <deftemplate-name> [extends <classname>] [<doc-comment>]
     [(slot <slot-name> [(default | default-dynamic <value>)]
                        [(type <typespec>)])]*)
```

The `<deftemplate-name>` is the head of the facts that will be created using this template. There may be an arbitrary number of slots. Each `<slot-name>` must be an atom. The `default` slot qualifier states that the default value of a slot in a new fact is given by `<value>`; the default is the atom `nil`. The 'default-dynamic' version will evaluate the given value each time a new fact using this template is asserted. The 'type' slot qualifier is accepted but not currently enforced by Jess; it specifies what data type the slot is allowed to hold. Acceptable values are ANY, INTEGER, FLOAT, NUMBER, ATOM, STRING, LEXEME, and OBJECT.

As an example, defining the following template:

```
Jess> (deftemplate automobile
    "A specific car."
    (slot make)
    (slot model)
    (slot year (type INTEGER))
    (slot color (default white)))
```

would allow you to define facts like this:

```
Jess> (assert (automobile (make Chrysler) (model LeBaron)
          (year 1997)))
  <Fact-0>
  Jess> (facts)
  f-0   (MAIN::automobile (make Chrysler) (model LeBaron)
                  (year 1997) (color white))
  For a total of 1 facts.
```

Note that the car is white by default. If you don't supply a default value for a slot, and then don't supply a value when a fact is asserted, the special value `nil` is used. Also note that any number of additional automobiles could also be simultaneously asserted onto the fact list using this template.

A given slot in a deftemplate fact can normally hold only one value. If you want a slot that can hold multiple values, use the `multislot` keyword instead:

```
Jess> (deftemplate box (slot location) (multislot contents))
  TRUE
  Jess>   (bind ?id (assert (box (location kitchen)
                  (contents spatula sponge frying-pan))))
  <Fact-1>
```

(We're saving the fact-id returned by (assert) in the variable ?id, for use below.) A multislot has the

default value () (the empty list) if no other default is specified.

You can change the values in the slots of an unordered fact using the `modify` command. Building on the immediately preceding example, we can move the box into the dining room:

```
Jess> (modify ?id (location dining-room))
  <Fact-1>
  Jess> (facts)
  f-0   (MAIN::automobile (make Chrysler) (model LeBaron)
                    (year 1997) (color white))
  f-1   (MAIN::box (location dining-room)
             (contents spatula sponge frying-pan))
  For a total of 2 facts.
```

The optional `extends` clause of the `deftemplate` construct lets you define one template in terms of another. For example, you could define a used-auto as a kind of automobile with more data:

```
Jess> (deftemplate used-auto extends automobile
    (slot mileage)
    (slot blue-book-value)
    (multislot owners))
  TRUE
```

A used-auto fact would now have all the slots of an automobile, plus three more. As we'll see later, this inheritance relationship will let you act on all automobiles (used or not) when you so desire, or only on the used ones.

Note that an ordered fact is very similar to an unordered fact with only one multislot. The similarity is so strong, that in fact this is how ordered facts are implemented in Jess. If you assert an ordered fact, Jess automatically generates a template for it. This generated template will contain a single slot named "__data". Jess treats these facts specially - the name of the slot is normally hidden when the facts are displayed. This is really just a syntactic shorthand, though; ordered facts really are just unordered facts with a single multislot named "__data".

2.7.3. The deffacts construct

Typing separate `assert` commands for each of many facts is rather tedious. To make life easier in this regard, Jess includes the `deffacts` construct. A `deffacts` construct is a simply a named list of facts. The facts in all defined `deffacts` are asserted into the knowledge base whenever a `reset` command is issued:

```
Jess> (deffacts my-facts "The documentation string"
    (foo bar)
    (box (location garage) (contents scissors paper rock))
    (used-auto (year 1992) (make Saturn) (model SL1)
               (mileage 120000) (blue-book-value 3500)
               (owners ejfried)))
  TRUE
  Jess> (reset)
  TRUE
```

```
Jess> (facts)
f-0   (MAIN::initial-fact)
f-1   (MAIN::foo bar)
f-2   (MAIN::box (location garage) (contents scissors paper rock))
f-3   (MAIN::used-auto (make Saturn) (model SL1) (year 1992)
                  (color white) (mileage 120000)
                  (blue-book-value 3500) (owners ejfried))
For a total of 4 facts.
```

Note that we can specify the slots of an unordered fact in any order (hence the name.) Jess rearranges our inputs into a *canonical order* so that they're always the same.

2.7.4. Definstance facts

You may have noticed that unordered facts look a bit like Java objects, or specifically, like Java Beans. The similarity is that both have a list of slots (for Java Beans, they're called *properties*) which contains values that might change over time. Jess has a mechanism for automatically generating templates that represent specific types of Java Beans. Jess can then use these templates to store a representation of a Java Bean's properties on the knowledge base. The knowledge base representation of the Bean can be *static* (changing infrequently, like a snapshot of the properties at one point in time) or dynamic (changing automatically whenever the Bean's properties change.) The Jess commands that make this possible are `defclass` and `definstance`. `defclass` tells Jess to generate a special template to represent a category of Beans, while `definstance` puts a representation of one specific Bean onto the fact base.

An example will probably help at this point. Let's say you have the following Java Bean class

```
import java.io.Serializable;
public class ExampleBean implements Serializable
{
  private String m_name = "Bob";
  public String getName() { return m_name; }
  public void setName(String s) { m_name = s; }
}
```

This Bean has one property called "name". Before we can insert any of these Beans onto the knowledge base, we need a template to represent them: we must use `defclass` to tell Jess to generate it:

```
Jess> (defclass simple ExampleBean)
  ExampleBean
  Jess> (ppdeftemplate simple)
  "(deftemplate MAIN::simple extends MAIN::__fact \"$JAVA-OBJECT$ ExampleBean\"
   (slot class (default <External-Address:jess.SerializablePD>))
   (slot name (default <External-Address:jess.SerializablePD>))
   (slot OBJECT (type 2048)))"
```

This is a strange looking template, but it does have a slot called "name", as we'd expect, that arises from the "name" property of our Bean. The slot "class" comes from the method `getClass()` that every object inherits from `java.lang.Object`, while the slot OBJECT is added by Jess; its value is always a reference to the Bean itself. See how the first argument to `defclass` is used as the template name.

Note that if you want your Java Beans to work with Jess's `bload` *and* `bsave` *commands, the individual classes need to implement the* `java.io.Serializable` *tag interface.*

Now let's say we want an actual ExampleBean in our knowledge base. Here we'll create one from Jess code, but it could come from anywhere. We will use the `definstance` function to add the object to the knowledge base.

```
Jess> (bind ?sb (new ExampleBean))
  <External-Address:ExampleBean>
  Jess> (definstance simple ?sb static)
  <Fact-0>
  Jess> (facts)
  f-0   (MAIN::simple (class <External-Address:java.lang.Class>)
                (name "Bob")
                (OBJECT <External-Address:ExampleBean>))
  For a total of 1 facts.
```

As soon as we issue the `definstance` command, a fact representing the Bean appears in the knowledge base.

Now watch what happens if we change the "name" property of our Bean.

```
Jess> (call ?sb setName "Fred")
  Jess> (facts)
  f-0   (MAIN::simple (class <External-Address:java.lang.Class>)
                (name "Bob")
                (OBJECT <External-Address:ExampleBean>))
  For a total of 1 facts.
```

Hmmm. The knowledge base still thinks our Bean's name is "Bob", even though we changed it to "Fred". What happens if we issue a `reset` command?

```
Jess> (reset)
  TRUE
  Jess> (facts)
  f-0   (MAIN::initial-fact)
  f-1   (MAIN::simple (class <External-Address:java.lang.Class>)
                (name "Fred")
                (OBJECT <External-Address:ExampleBean>))
  For a total of 2 facts.
```

`reset` updates the definstance facts in the knowledge base to match their Java Beans. This behaviour is what you get when (as we did here) you specify *static* in the `definstance` command. Static definstances are refreshed only when a reset is issued.

If you want to have your definstance facts stay continuously up to date, Jess needs to be notified whenever a Bean property changes. For this to happen, the Bean has to support the use of `java.beans.PropertyChangeListener`s. For Beans that fulfill this requirement, you can specify *dynamic* in the definstance command, and the knowledge base will be updated every time a property of the Bean changes. Jess comes with some example Beans that can be used in this way; see, for example, the

`Jess61RC1/jess/examples/simple` directory.

`defclass`es, like deftemplates, can `extend` one another. In fact, deftemplates can extend defclasses, and defclasses can extend deftemplates. Of course, for a `defclass` to extend a `deftemplate`, the corresponding Bean class must have property names that match the template's slot names. Note, also, that just because two Java classes have an inheritance relationship doesn't mean that if both are `defclass`ed the two `defclass`es will. You must explicitly declare all such relationships using `extends.` See the full documenation for `defclass` for details.

One final note about Java Beans used with Jess: Beans are often operating in a multithreaded environment, and so it's important to protect their data with synchronized blocks or synchronized methods. However, sending `PropertyChangeEvents` while holding a lock on the Bean itself can be dangerous, as the Java Beans Specification points out:

> "In order to reduce the risk of deadlocks, we strongly recommend that event sources should avoid holding their own internal locks when they call event listener methods. Specifically, as in the example code in Section 6.5.1, we recommend they should avoid using a synchronized method to fire an event and should instead merely use a synchronized block to locate the target listeners and then call the event listeners from unsynchronized code." -- JavaBean Specification, v 1.0.1, p.31.

Failing to heed this advice can indeed cause deadlocks in Jess.

2.8. Defrules

Now that we've learned how to develop a knowledge base, we can answer the obvious question: what is it good for? The answer is that queries can search it to find relationships between facts, and rules can take actions based on the contents of one or more facts.

A Jess rule is something like an `if... then` statement in a procedural language, but it is not used in a procedural way. While `if... then` statements are executed at a specific time and in a specific order, according to how the programmer writes them, Jess rules are executed whenever their `if` parts (their *left-hand-sides* or *LHS*s) are satisfied, given only that the rule engine is running. This makes Jess rules less deterministic than a typical procedural program. See the chapter on the Rete algorithm for an explanation of why this architecture can be many orders of magnitude faster than an equivalent set of traditional `if... then` statements.

Rules are defined in Jess using the `defrule` construct. A very simple rule looks like this:

```
Jess> (defrule do-change-baby
    "If baby is wet, change baby's diaper."
    (baby-is-wet)
    =>
    (change-baby))
```

This rule has two parts, separated by the "=>" symbol (which you can read as "then".) The first part consists of the LHS *pattern* `(baby-is-wet)`. The second part consists of the RHS *action* `(change-baby).` Although it's hard to tell due to the LISP-like syntax, the LHS of a rule consists of patterns which are used to match facts in the knowledge base, while the RHS contains function calls.

*The LHS of a rule (the "if" part) consists of patterns that match facts, **NOT** function calls. The actions of a rule (the "then" clause) are made up of function calls. The following rule does **NOT** work:*

```
Jess> (defrule wrong-rule
      (eq 1 1)
      =>
      (printout t "Just as I thought,  1 == 1!" crlf))
```

*This rule will NOT fire just because the function call (eq 1 1) would evaluate to true. Instead, Jess will try to find a fact on the knowledge base that looks like (eq 1 1). Unless you have previously asserted such a fact, this rule will **NOT** be activated and will not fire. If you want to fire a rule based on the evaluation of a function, you can use the test CE.*

Our example rule, then, will be activated when the fact `(baby-is-wet)` appears in the knowledge base. When the rule executes, or *fires*, the function `(change-baby)` is called (presumably this function is defined elsewhere in our imaginary program.) Let's turn this rule into a complete program. The function `watch all` tells Jess to print some useful diagnostics as we enter our program.

```
Jess> (watch all)
  TRUE
  Jess>  (reset)
  ==> f-0 (MAIN::initial-fact)
  TRUE
  Jess> (deffunction change-baby () (printout t "Baby is now dry" crlf))
  TRUE
  Jess> (defrule do-change-baby
      (baby-is-wet)
      =>
      (change-baby))
  do-change-baby: +1+1+1+t
  TRUE
  Jess>  (assert (baby-is-wet))
   ==> f-1 (MAIN::baby-is-wet)
  ==> Activation: MAIN::do-change-baby :  f-1
  <Fact-1>
```

Some of these diagnostics are interesting. We see first of all how issuing the `reset` command asserts the fact `(initial-fact)`. You should always issue a `reset` command when working with rules. When the rule itself is entered, we see the line "+1+1+t". This tells you something about how the rule is interpreted by Jess internally (see The Rete Algorithm for more information.) When the fact `(baby-is-wet)` is asserted, we see the diagnostic "Activation: MAIN::do-change-baby : f-1". This means that Jess has noticed that the rule `do-change-baby` has all of its LHS conditions met by the given list of facts ("f-1").

After all this, our rule didn't fire; why not? Jess rules only fire while the rule engine is running (although they can be *activated* while the engine is not running.) To start the engine running, we issue the `run` command.

```
Jess> (run)
  FIRE 1 MAIN::do-change-baby f-1
  Baby is now dry
  <== Focus MAIN
  1
```

As soon as we enter the `run` command, the activated rule fires. Since we have `watch all`, Jess prints the diagnostic `FIRE 1 do-change-baby f-1` to notify us of this. We then see the output of the rule's RHS actions. The final number "1" is the number of rules that fired (it is the return value of the `run` command.) The `run` function returns when there are no more activated rules to fire.

What would happen if we entered `(run)` again? Nothing. A rule will be activated only once for a given set of facts; once it has fired, that rule will not fire again for the same list of facts. We won't change the baby again until the `(baby-is-wet)` fact is retracted (perhaps by `(reset)` and asserted again. In fact, this rule should itself retract the `(baby-is-wet)` fact itself; to learn how, see the section on pattern bindings, below.

Rules are uniquely identified by their name. If a rule named `my-rule` exists, and you define another rule named `my-rule,` the first version is deleted and will not fire again, even if it was activated at the time the new version was defined.

2.8.1. Basic Patterns

If all the patterns of a rule had to be given literally as above, Jess would not be very powerful. However, patterns can also include wildcards and various kinds of *predicates* (comparisons and boolean functions). You can specify a variable name instead of a value for a field in any of a rule's patterns (but not the pattern's head). A variable matches any value in that position within a rule. For example, the rule:

```
Jess> (defrule example-2
    (a ?x ?y)
    =>
    (printout t "Saw 'a " ?x " " ?y "'" crlf))
```

will be activated each time any fact with head `a` having two fields is asserted: `(a b c)`, `(a 1 2)`, `(a a a)`, and so forth. As in the example, the variables thus matched in the patterns (or LHS) of a rule are available in the actions (RHS) of the same rule.

Each such variable field in a pattern can also include any number of tests to qualify what it will match. Tests follow the variable name and are separated from it and from each other by ampersands (&) or pipes (|). (The variable name itself is actually optional.) Tests can be:

- A literal value (in which case the variable matches *only* that value); for example, the values `b` and `c` in `(a b c)`.
- Another variable (which must have been matched earlier in the rule's LHS). This will constrain the field to contain the same value as the variable was first bound to; for example, `(a ?X ?X)` will only match "a" facts followed by two equal values.
- A colon (`:`) followed by a function call, in which case the test succeeds if the function returns the special value `TRUE`. These are called *predicate constraints*; for example, `(a ?X&:(> ?X 10)`

matches "a" facts with one field, a number greater than 10.

- An equals sign (=) followed by a function call. In this case the field must match the return value of the function call. These are called *return value constraints*. Note that both predicate constraints and return-value constraints can refer to variables bound elsewhere in this or any preceding pattern in the same rule. ***Note:*** pretty-printing a rule containing a return value contstraint will show that it has been transformed into an equivalent predicate constraint. An example of a return-value constraint would be (a ?X =(+ ?X 1)), which matches "a" facts with two fields, both numbers with the second number greater than the first by one.
- Any of the other options preceded by a tilde (~), in which case the sense of the test is reversed (inequality or false); for example `(a ?X ~?X)` matches "a" facts with two fields as long as the two fields contains different values.

Ampersands (&) represent logical "and", while pipes (|) represent logical "or." & has a higher precedence than |, so that the following

```
(foo ?X&:(oddp ?X)&:(< ?X 100)|0)
```

matches a `foo` fact with a single field containing either an odd number less than 100, or 0.

Here's an example of a rule that uses several kinds of tests:

```
Jess> (defrule example-3
    (not-b-and-c ?n1&~b ?n2&~c)
    (different ?d1 ?d2&~?d1)
    (same ?s ?s)
    (more-than-one-hundred ?m&:(> ?m 100))
    (red-or-blue red|blue)
    =>
    (printout t "Found what I wanted!" crlf))
```

The first pattern will match a fact with head `not-b-and-c` with exactly two fields such that the first is not `b` and the second is not `c`. The second pattern will match any fact with head `different` and two fields such that the two fields have different values. The third pattern will match a fact with head `same` and two fields with identical values. The fourth pattern matches a fact with head `more-than-one-hundred` and a single field with a numeric value greater than 100. The last pattern matches a fact with head `red-or-blue` followed by either the atom `red` or the atom `blue`.

A few more details about patterns: you can match a field without binding it to a variable by omitting the variable name and using just a question mark (?) as a placeholder. You can match any number of fields in a multislot or unordered fact using a multivariable (one starting with $?):

```
Jess> (defrule example-4
    (grocery-list $?list)
    =>
    (printout t "I need to buy " $?list crlf))
  TRUE
  Jess> (assert (grocery-list eggs milk bacon))
  <Fact-0>
  Jess> (run)
  I need to buy (eggs milk bacon)
  1
```

If you match to a defglobal with a pattern like (foo ?*x*), the match will only consider the value of the defglobal when the fact is asserted. Subsequent changes to the defglobal's value will *not* invalidate the match - i.e., the match does not reflect the current value of the defglobal, but only the value at the time the matching fact was asserted.

2.8.2. Pattern bindings

Sometimes you need a handle to an actual fact that helped to activate a rule. For example, when the rule fires, you may need to retract or modify the fact. To do this, you use a pattern-binding variable:

```
Jess> (defrule example-5
    ?fact <- (a "retract me")
    =>
    (retract ?fact))
```

The variable (`?fact`, in this case) is bound to the particular fact that activated the rule.

Note that ?fact is a jess.Value object of type RU.FACT, not an integer. It is basically a reference to a jess.Fact object. You can convert an ordinary number into a FACT using the `fact-id` function. You can convert a FACT into an integer when necessary by using reflection to call the `Fact.getFactId()` function. The `jess.Value.factValue()` method can be called on a FACT Value to obtain the actual `jess.Fact` object from Java code. In Jess code, a fact-id essentially *is* a `jess.Fact`, and you can call `jess.Fact` methods on a fact-id directly:

```
Jess> (defrule example-5-1
  ?fact <- (initial-fact)
  =>
  (printout t (call ?fact getName) crlf))
  TRUE
  Jess> (reset)
  TRUE
  Jess> (run)
  initial-fact
  1
```

See the section on the jess.FactIDValue class for more information.

Note that once a fact is asserted, Jess will always use the same `jess.Fact` object to represent it, even if the original fact is modified. Therefore, you can store references to fact objects in the slots of other facts as a way of representing structured data.

2.8.3. Salience and conflict resolution

Each rule has a property called *salience* that is a kind of rule priority. Activated rules of the highest salience will fire first, followed by rules of lower salience. To force certain rules to always fire first or last, rules can include a salience declaration:

```
Jess> (defrule example-6
    (declare (salience -100))
    (command exit-when-idle)
    =>
    (printout t "exiting..." crlf))
```

Declaring a low salience value for a rule makes it fire after all other rules of higher salience. A high value makes a rule fire before all rules of lower salience. The default salience value is zero. Salience values can be integers, global variables, or function calls. See the `set-salience-evaluation` command for details about when such function calls will be evaluated.

The order in which multiple rules of the same salience are fired is determined by the active *conflict resolution strategy*. Jess comes with two strategies: "depth" (the default) and "breadth." In the "depth" strategy, the most recently activated rules will fire before others of the same salience. In the "breadth" strategy, rules fire in the order in which they are activated. In many situations, the difference does not matter, but for some problems the conflict resolution strategy is important. You can write your own strategies in Java; see the chapter on extending Jess with Java for details. You can set the current strategy with the `set-strategy` command.

Note that the use of salience is generally discouraged, for two reasons: first it is considered bad style in rule-based programming to try to force rules to fire in a particular order. Secondly, use of salience will have a negative impact on performance, at least with the built-in conflict resolution strategies.

You can see the list of activated, but not yet fired, rules with the `agenda` command.

2.8.4. The ’and’ conditional element.

Any number of patterns can be enclosed in a list with `and` as the head. The resulting pattern is matched if and only if all of the enclosed patterns are matched. By themselves, `and` groups aren’t very interesting, but combined with or and not conditional elements, they can be used to construct complex logical conditions.

The entire left hand side of every rule and query is implicitly enclosed in an `and` conditional element.

2.8.5. The ’or’ conditional element.

Any number of patterns can be enclosed in a list with `or` as the head. The resulting pattern is matched if one or more of the patterns inside the `or` are matched. If more than one of the subpatterns are matched, the `or` is matched more than once:

```
Jess> (defrule or-example-1
     (or (a) (b) (c))
     =>)
  Jess> (assert (a) (b) (c))
  Jess> (printout t (run) crlf)
  3
```

An `and` group can be used inside of an `or` group, and vice versa. In the latter case, Jess will rearrange the patterns so that there is a single `or` at the top level. For example, the rule

```
(defrule or-example-2a
    (and (or (a)
             (b))
         (c))
    =>)
```

will be automatically rearranged to

```
(defrule or-example-2b
    (or (and (a) (c))
        (and (b) (c)))
    =>)
```

DeMorgan's second rule of logical equivalence, namely

```
(not (or (x) (y)))  => (and (not (x)) (not (y)))
```

will be used when necessary to hoist an `or` up to the top level.

Note that if the right hand side of a rule uses a variable defined by matching on the left hand side of that rule, and the variable is defined by one or more branches of an `or` pattern but not all branches, then a runtime error may occur.

2.8.5.1. Subrule generation and the 'or' conditional element.

A rule containing an 'or' conditional element with *n* branches is precisely equivalent to *n* rules, each of which has one branch as its left hand side. In fact, this is how the `or` conditional element is implemented: Jess imternally generates one new rule for each branch. Each of these generated rules is a *subrule*. For a rule named *rule-name*, the first subrule is named `rule-name`, the second is `rule-name&1`, the third is `rule-name&2`, etc. Each of these subrules is added to the Rete network individually. If you execute the (rules) command, you will see each of them listed separately. If you use the `ppdefrule` function to see a pretty-print representation of a subrule, you will see only the representation of that rule. Note that since '&' is a token delimiter in the Jess grammar, you can only refer to a subrule with an ampersand in the name by placing the whole name in quotes; i.e., `(ppdefrule "rule-name&6")`.

Jess knows that the subrules created from a given rule are related. If a rule is removed (either using `undefrule` or implicitly by defining a new rule with the same name as an existing one) every subrule associated with that rule is undefined.

Regarding subrules and efficiency: remember that similar patterns are shared between rules in the Rete network. Therefore, splitting a rule into subrules does *not* mean that the amount of pattern-matching work is increased; much of the splitting may indeed be undone when the rules are compiled into the network.

On the other hand, keep the implementation in mind when you define your rules. If an `or` conditional element is the first pattern on a rule, all the subsequent pattern-matching on that rule's left-hand side won't be shared between the subrules, since sharing only occurs as far as two rules are similar reading from the top down. Placing `or` conditional elements near the end of a rule will lead to more sharing between the subrules.

> *Note: although subrules will probably always be part of the implementation of the `or` conditional element in Jess, it is very likely that they will no longer be user-visible at some time in the future.*

2.8.6. The 'not' conditional element.

Any single pattern can be enclosed in a list with `not` as the head. In this case, the pattern is considered to match if a fact (or set of facts) which matches the pattern is *not* found. For example:

```
Jess> (defrule example-7
     (person ?x)
     (not (married ?x))
     =>
     (printout t ?x " is not married!" crlf))
```

Note that a `not` pattern cannot define any variables that are used in subsequent patterns (since a `not` pattern does not match any facts, it cannot be used to define the values of any variables!) You can introduce variables in a `not` pattern, so long as they are used only within that pattern; i.e,

```
Jess> (defrule no-odd-numbers
     (not (number ?n&:(oddp ?n)))
     =>
     (printout t "There are no odd numbers." crlf))
```

Similarly, a `not` pattern can't have a pattern binding.

A `not` CE is evaluated only when either a fact matching it exists, or when the pattern immediately before the `not` on the rule's LHS is evaluated. If a `not` CE is the first pattern on a rule's LHS, or is the the first the pattern in an `and` group, or is the only pattern on a given branch of an `or` group, the pattern `(initial-fact)` is inserted to become this important preceding pattern. Therefore, the fact `(initial-fact)` created by the `reset` command is necessary to the proper functioning of some `not` patterns. For this reason, it is especially important to issue a `reset` command before attempting to run the rule engine when working with `not` patterns.

Multiple `not` CEs can be nested to produce some interesting effects (see the discussion of the `exists` CE).

The `not` CE can be used in arbitrary combination with the and and or CEs. You can define complex logical structures this way. For example, suppose you want a rule to fire once if for every fact (a ?x), there is a fact (b ?x). You could express that as

```
Jess> (defrule forall-example
    (not (and (a ?x) (not (b ?x))))
    =>)
```

i.e., "It is not true that for some ?x, there is an (a ?x) and no (b ?x)". You might recognize this as the CLIPS `forall` conditional element; a future version of Jess will include the `forall` shorthand.

2.8.7. The 'test' conditional element.

A pattern with `test` as the head is special; the body consists not of a pattern to match against the knowledge base but of one or more boolean functions, which are evaluated in order. The results determine whether the pattern matches. A `test` pattern fails if and only if one of the functions evaluates to the atom `FALSE`; if they all evaluate to `TRUE` or any other value, the pattern with "match." For example:

```
Jess> (deftemplate person (slot age))
  Jess> (defrule example-8
     (person (age ?x))
     (test (> ?x 30))
     =>
     (printout t ?x " is over 30!" crlf))
```

Short-circuit evaluation is used; i.e., if a function call evaluates to `FALSE`, no further functions are evaluated and the `test` CE fails immediately. Note that a `test` pattern, like a `not`, cannot contain any variables that are not bound before that pattern. `test` and `not` may be combined:

```
  (not (test (eq ?X 3)))
```

is equivalent to:

```
  (test (neq ?X 3))
```

A `test` CE is evaluated every time the *preceding* pattern on the rule's LHS is evaluated. Therefore the following two rules are precisely equivalent in behaviour:

```
Jess> (defrule rule_1
    (foo ?X)
    (test (> ?X 3))
    =>)

  Jess> (defrule rule_2
    (foo ?X&:(> ?X 3))
    =>)
```

For rules in which a `test` CE is the first pattern on the LHS or the first pattern in a branch of an `or` CE, the pattern `(initial-fact)` is inserted to become the "preceding pattern" for the `test`. The fact `(initial-fact)` is therefore also important for the proper functioning of the `test` conditional element; the caution about `reset` in the preceding section applies equally to `test.`

2.8.7.1. Time-varying method returns

One useful property of the `test` CE is that it's the only valid place to put tests whose results might change without the contents of any slot changing. For example, imagine that you've got two Java class, `A` and `B`, and that `A` has a method `contains` which takes a `B` as an argument and returns boolean. Further, imagine that for any given `B` object, the return value of `contains` will change over time. Finally, imagine that you've defclasses both these classes and are writing rules to work with them. Under these circumstances, a set of patterns like this:

```
(A (OBJECT ?a))
(B (OBJECT ?b&:(?a contains ?b)))
```

is incorrect. If the return value of contains changes, the match will be invalidated and Jess's internal data structures may be corrupted. In particular, this kind of construct tends to cause memory leaks.

The correct way to express this same set of patterns is to use the `test` conditional element, like this:

```
(A (OBJECT ?a))
(B (OBJECT ?b))
(test (?a contains ?b))
```

The function `contains` is now guaranteed to be called at most once for each combination of target and argument, and so any variation in return value will have no impact.

2.8.8. The 'logical' conditional element.

The `logical` conditional element lets you specify *logical dependencies* among facts. All the facts asserted on the RHS of a rule become dependent on the matches to the `logical`patterns on that rule's LHS. If any of the matches later become invalid, the dependent facts are retracted automatically. In this simple example, a single fact is made to depend on another single fact:

```
Jess> (defrule rule-1
    (logical (faucet-open))
    =>
    (assert (water-flowing)))
  TRUE
  Jess> (assert (faucet-open))
  <Fact-0>
  Jess> (run)
  1
  Jess> (facts)
  f-0   (MAIN::faucet-open)
  f-1   (MAIN::water-flowing)
  For a total of 2 facts.
  Jess> (watch facts)
  TRUE
  Jess> (retract (fact-id 0))
  <== f-0 (MAIN::faucet-open)
  <== f-1 (MAIN::water-flowing)
  TRUE
```

The `(water-flowing)` fact is logically dependent on the `(faucet-open)` fact, so when the latter is

retracted, the former is removed, too.

A fact may receive logical support from multiple sources -- i.e., it may be asserted multiple times with a different set of logical supports each time. Such a fact isn't automatically retracted unless each of its logical supports is removed.

If a fact is asserted without explicit logical support, it is said to be *unconditionally supported.* If an unconditionally supported fact also receives explicit logical support, removing that support will not cause the fact to be retracted.

If one or more `logical` CEs appear in a rule, they must be the first patterns in that rule; i.e., a `logical` CE cannot be preceded in a rule by any other kind of CE.

Definstance facts are no different than other facts with regard to the `logical` CE. Definstance facts can provide logical support and can receive logical support. In the current implementation, definstance facts can only provide logical support as a whole. In a future version of Jess, it will be possible for a definstance fact to provide logical support based on any combination of individual slot values.

The `logical` CE can be used together with all the other CEs, including `not` and `exists`. A fact can thus be logically dependent on the non-existence of another fact, or on the existence of some category of facts in general.

2.8.9. The 'unique' conditional element.

The `unique` CE has been removed. The parser will accept but ignore it.

2.8.10. The 'exists' conditional element.

A pattern can be enclosed in a list with `exists` as the head. An `exists` CE is true if there exist any facts that match the pattern, and false otherwise. `exists` is useful when you want a rule to fire only once, although there may be many facts that could potentially activate it.

```
Jess> (defrule exists-demo
    (exists (honest ?))
    =>
    (printout t "There is at least one honest man!" crlf))
```

If there are any honest men in the world, the rule will fire once and only once.

`exists` may not be combined in the same pattern with a `test` CE.

Note that `exists` is precisely equivalent to (and in fact, is implemented as) two nested `not` CEs; i.e., `(exists (A))` is the same as `(not (not (A))).`

2.8.11. Node index hash value.

The *node index hash value* is a tunable performance-related parameter that can be set globally or on a per-rule basis. A small value will save memory, possibly at the expense of performance; a larger value will use more memory but lead to faster rule LHS execution.

In general, you might want to declare a large value for a rule that was likely to generate many partial matches (prime numbers are the best choices:)

```
Jess> (defrule nihv-demo
      (declare (node-index-hash 169))
      (item ?a)
      (item ?b)
      (item ?c)
      (item ?d)
      =>)
```

See the discussion of the `set-node-index-hash` function for a full discussion of this value and what it means.

2.8.12. Forward and backward chaining

The rules we've seen so far have been *forward-chaining* rules, which basically means that the rules are treated as `if... then` statements, with the engine passively executing the RHSs of activated rules. Some rule-based systems, notable Prolog and its derivatives, support *backward chaining.* In a backwards chaining system, rules are still `if... then` statements, but the engine seeks steps to activate rules whose preconditions are not met. This behaviour is often called "goal seeking". Jess supports both forward and backward chaining. Note that the explanation of backward chaining in Jess is necessarily simplified here since full explanation requires a good understanding of the underlying algorithms used by Jess.

To use backward chaining in Jess, you must first declare that certain fact templates will be *backward chaining reactive* using the `do-backward-chaining` function:

```
Jess> (do-backward-chaining factorial)
```

If the template is unordered -- i.e., if it is explicitly defined with a (deftemplate) construct -- then it must be defined *before* calling `do-backward-chaining`. Then you can define rules which match such patterns. Note that do-backward-chaining must be called *before* defining any rules which use the template.

```
Jess> (defrule print-factorial-10
    (factorial 10 ?r1)
    =>
    (printout t "The factorial of 10 is " ?r1 crlf))
```

When the rule compiler sees that a pattern matches a backward chaining reactive template, it rewrites the rule and inserts some special code into the internal representation of the rule's LHS. This code asserts a fact onto the fact-list that looks like

```
  (need-factorial 10 nil)
```

if, when the rule engine is reset, there are no matches for this pattern. The head of the fact is constructed by taking the head of the reactive pattern and adding the prefix "need-".

Now, you can write rules which match these need-(x) facts.

```
Jess> (defrule do-factorial
    (need-factorial ?x ?)
    =>
    (bind ?r 1)
    (bind ?n ?x)
    (while (> ?n 1)
      (bind ?r (* ?r ?n))
      (bind ?n (- ?n 1)))
    (assert (factorial ?x ?r)))
```

The rule compiler rewrites rules like this too: it adds a negated match for the factorial pattern itself to the rule's LHS.

The end result is that you can write rules which match on (factorial), and if they are close to firing except they need a (factorial) fact to do so, any (need-factorial) rules may be activated. If these rules fire, then the needed facts appear, and the (factorial)-matching rules fire. This, then, is backwards chaining! Jess will chain backwards through any number of reactive patterns. For example:

```
Jess> (do-backward-chaining foo)
  TRUE
  Jess> (do-backward-chaining bar)
  TRUE
  Jess> (defrule rule-1
    (foo ?A ?B)
    =>
    (printout t foo crlf))
  TRUE
  Jess> (defrule create-foo
    (need-foo $?)
    (bar ?X ?Y)
    =>
    (assert (foo A B)))
  TRUE
  Jess> (defrule create-bar
    (need-bar $?)
    =>
    (assert (bar C D)))
  TRUE
  Jess> (reset)
  TRUE
  Jess> (run)
  foo
  3
```

In this example, none of the rules can be activated at first. Jess sees that `rule-1` could be activated if there were an appropriate `foo` fact, so it generates the request `(need-foo nil nil)`. This matches

part of the LHS of rule `create-foo` cannot fire for want of a `bar` fact. Jess therefore creates a `(need-bar nil nil)` request. This matches the LHS of the rule `create-bar`,which fires and asserts `(bar C D)`. This activates `create-foo,` which fires, asserts `(foo A B)`, thereby activating `rule-1`, which then fires.

There is a special conditional element, `(explicit),` which you can wrap around a pattern to inhibit backwards chaining on an otherwise reactive pattern.

2.9. Defqueries

The `defquery` construct lets you create a special kind of rule with no right-hand-side. While rules act spontaneously, queries are used to search the knowledge base under direct program control. A rule is activated once for each matching set of facts, while a query gives you a `java.util.Iterator` of all the matches. An example should make this clear. Suppose we have defined this query:

```
Jess> (defquery search
    "Finds foo facts with a specified first field"
    (declare (variables ?X))
    (foo ?X ?Y))
```

Then if the knowledge base contains these facts:

```
Jess> (deffacts data
     (foo blue red)
     (bar blue green)
     (foo blue pink)
     (foo red blue)
     (foo blue blue)
     (foo orange yellow)
     (bar blue purple))
```

Then the following Jess code Will print the output shown:

```
Jess> (reset)
     Jess> (bind ?it (run-query search blue))

     Jess> (while (?it hasNext)
            (bind ?token (call ?it next))
            (bind ?fact (call ?token fact 1))
            (bind ?slot (fact-slot-value ?fact __data))
            (bind ?datum (nth$ 2 ?slot))
            (printout t ?datum crlf))

     red
     pink
     blue
     FALSE
```

because these three values follow `blue` in a `foo` fact.

Let's break this code down to see what it's doing. As previously stated, `(run-query)` returns the query results as a `java.util.Iterator`. The `Iterator` interface has a method `next()` that you call to retrieve each individual result; it also has a `hasNext()` method which returns true as long as there are more results to return. That explains the `(while (?it hasNext) ... (call ?it next))` structure.

Each individual result is a jess.Token object. A token is basically just a collection of jess.Fact objects; here it is a collection that matches this query. We call the `fact()` method of `jess.Token` to retrieve the individual facts within the Token. Note that each match begins with an extra fact - a `__query-trigger` fact that triggers the matching process, asserted by the run-query command; hence the argument to the call to `Token.fact()` above is 1, not 0.

Once we have the right fact, we're interested in the second item in the data part of the fact (the first item in the fact is the *head* and it's stored differently.) As stated above, the slot data for ordered facts is stored in a single multifield slot named `__data`. We retrieve the contents of that slot using the `fact-slot-value` function, then use the `nth$` function to retrieve the second slot (`nth$` uses a one-based index.)

The following Java code is similar to the Jess snippets above. It defines the same query and deffacts, runs the query and then collects the `red, pink` and `blue` values in a Vector as Strings.

```
import jess.*;
import java.util.*;

public class ExQuery
{
  public static void main(String [] argv) throws JessException
  {
    // Create engine, define query and data
    Rete r = new Rete();
    r.executeCommand("(defquery search (declare (variables ?X)) (foo ?X ?Y))");
    r.executeCommand("(deffacts data" +
                     "(foo blue red)" +
                     "(bar blue green)" +
                     "(foo blue pink)" +
                     "(foo red blue)" +
                     "(foo blue blue)" +
                     "(foo orange yellow)" +
                     "(bar blue purple))");

    // Assert all the facts
    r.reset();
    // Run the query, store the result
    r.store("RESULT", r.runQuery("search",
                     new ValueVector().add(new Value("blue", RU.ATOM))));
    r.executeCommand("(store RESULT (run-query search blue))");

    // Fetch the result (an Iterator).
    Iterator e = (Iterator) r.fetch("RESULT").externalAddressValue(null);
    ArrayList v = new ArrayList();

    // Pick each element of the Iterator apart and store the
```

```
    // interesting part in the ArrayList v.
    while (e.hasNext())
    {
      Token t = (Token) e.next();

      // We want the second fact in the token - the first is the query trigger
      Fact f = t.fact(1);

      // The first and only slot of this fact is the __data multislot.
      ValueVector multislot = f.get(0).listValue(null);

      // The second element of this slot is the datum we're interested in.
      v.add(multislot.get(1).stringValue(null));
    }
    for (Iterator answers = v.iterator(); answers.hasNext();)
      System.out.println(answers.next());
  }
}
C:\> java ExQuery
    red
  pink
  blue
```

Defqueries can use virtually all of the same features that rule LHSs can, except for salience. You can use the `node-index-hash` declaration, just as for rules.

2.9.1. The variable declaration

You might have already realized that two different kinds of variables can appear in a query: those that are "internal" to the query, like `?Y` in the query above, and those that are "external", or to be specified in the `run-query` command when the query is executed. Jess assumes all variables in a query are internal by default; you must declare any external variables explicitly using the syntax

```
(declare (variables ?X ?Y ...))
```

which is quite similar to the syntax of a rule salience declaration.

2.9.2. The max-background-rules declaration

It can be convenient to use queries as triggers for backward chaining. For this to be useful, `Rete.run()` must be called while the query is being evaluated, to allow the backward chaining to occur. Facts generated by rules fired during this run may appear as part of the query results. (If this makes no sense whatsoever to you, don't worry about it; just skip over this section for now.)

By default, no rules will fire while a query is being executed. If you want to allow backward chaining to occur in response to a query, you can use the `max-background-rules` declaration -- i.e.,

```
(declare (max-background-rules 10))
```

would allow a maximum of 10 rules to fire while this particular query was being executed.

2.9.3. The run-query command

The `run-query` command lets you supply values for the external variables of a query and obtain a list of matches. This function returns a `java.util.Iterator` of jess.Token object, one for each matching combination of facts. The example code above calls `fact(0)` on each `jess.Token`, to get the first `jess.Fact` object from the `jess.Token,` then calls `get(0)` on the fact to get the data from the first slot (which for ordered facts, is a multislot named __data; see the documentation for jess.Fact) and then uses `(nth$ 2)`to get the second entry in that multislot.

Note that each token will contain one more fact than there are patterns on the query's LHS; this extra fact is used internally by Jess to execute the query.

You must supply exactly one value for each external variable of the named query.

2.9.4. The count-query-results command

To obtain just the number of matches for a query, you can use the `count-query-results` function. This function accepts the same arguments as run-query, but returns an integer, the number of matches.

2.9.5. The future of queries

`defquery` is a new feature, and the syntax may change; in particular, a simpler mechanism for obtaining query results may be defined. Suggestions are welcome.

2.10. Defmodules

A typical rule-based system can easily include hundreds of rules, and a large one can contain many thousands. Developing such a complex system can be a difficult task, and preventing such a multitude of rules from interfering with one another can be hard too.

You might hope to mitigate the problem by partitioning a rule base into manageable chunks. *Modules* let you divide rules and templates into distinct groups. The commands for listing constructs let you specify the name of a module, and can then operate on one module at a time. If you don't explicitly specify a module, these commands (and others) operate by default on the *current module.* If you don't explicitly define any modules, the current module is always the *main module,* which is named MAIN. All the constructs you've seen so far have been defined in MAIN, and therefore are often preceded by "MAIN::" when displayed by Jess.

Besides helping you to manage large numbers of rules, modules also provide a control mechanism: the rules in a module will fire only when that module has the *focus*, and only one module can be in focus at a time.

> ***Note for CLIPS users:*** Jess's `defmodule` construct is similar to the CLIPS construct by the same name, but it is not identical. The syntax and the name resolution mechanism are simplified. The focus mechanism is much the same.

2.10.1. Defining constructs in modules

You can define a new module using the `defmodule` construct:

```
Jess> (defmodule WORK)
    TRUE
```

You can place a deftemplate, defrule, or deffacts into a specific module by qualifying the name of the construct with the module name:

```
Jess> (deftemplate WORK::job (slot salary))
    TRUE
  Jess> (list-deftemplates WORK)
    WORK::job
    For a total of 1 deftemplates.
```

Once you have defined a module, it becomes the *current module:*

```
Jess> (get-current-module)
    MAIN
  Jess> (defmodule COMMUTE)
    TRUE
  Jess> (get-current-module)
    COMMUTE
```

If you don't specify a module, all deffacts, templates and rules you define will automatically become part of the current module:

```
Jess> (deftemplate bus (slot route-number))
    TRUE
  Jess> (defrule take-the-bus
  ?bus <- (bus (route-number 76))
  (have-correct-change)
  =>
  (get-on ?bus))
    TRUE
  Jess> (ppdefrule take-the-bus)
    "(defrule COMMUTE::take-the-bus
       ?bus <- (COMMUTE::bus (route-number 76))
       (COMMUTE::have-correct-change)
       =>
       (get-on ?bus))"
```

You can set the current module explicitly using the `set-current-module` function. Note that the implied template `have-correct-change` was created in the COMMUTE module, because that's where the rule was defined.

2.10.2. Modules, scope, and name resolution

A module defines a *namespace* for templates and rules. This means that two different modules can each contain a rule with a given name without conflicting -- i.e., rules named `MAIN::initialize` and `COMMUTE::initialize` could be defined simultaneously and coexist in the same program. Similarly, the templates `COMPUTER::bus` and `COMMUTE::bus` could both be defined. Given this fact, there is the question of how Jess decides which template the definition of a rule or query is referring to.

When Jess is compiling a rule or deffacts definition, it will look for templates in three places, in order:

1. If a pattern explicitly names a module, only that module is searched.
2. If the pattern does not specify a module, then the module in which the rule is defined is searched first.
3. If the template is not found in the rule's module, the module MAIN is searched last. Note that this makes the MAIN module a sort of global namespace for templates.

The following example illustrates each of these possibilities:

```
Jess> (assert (MAIN::mortgage-payment 2000))
   <Fact-0>
  Jess> (defmodule WORK)
    TRUE
  Jess> (deftemplate job (slot salary))
    TRUE
  Jess> (defmodule HOME)
    TRUE
  Jess> (deftemplate hobby (slot name) (slot income))
    TRUE
  Jess> (defrule WORK::quit-job
    (job (salary ?s))
    (HOME::hobby (income ?i&:(> ?i (/ ?s 2))))
    (mortgage-payment ?m&:(< ?m ?i))
    =>
    (call-boss)
    (quit-job))
    TRUE
  Jess> (ppdefrule WORK::quit-job)
    "(defrule WORK::quit-job
     (WORK::job (salary ?s))
     (HOME::hobby (income ?i&:(> ?i (/ ?s 2))))
     (MAIN::mortgage-payment ?m&:(< ?m ?i))
     =>
     (call-boss)
     (quit-job))"
```

In this example, three deftemplates are defined in three different modules: `MAIN::mortgage-payment`, `WORK::job`, and `HOME::hobby`. Jess finds the WORK::job template because the rule is defined in the WORK module. It finds the HOME::hobby template because it is explicitly qualified with the module name. And the MAIN::mortgage-payment template is found because the MAIN module is always searched as a last resort if no module name is specified.

Commands which accept the name of a construct as an argument (like `ppdefrule`, `ppdeffacts`, etc) will search for the named construct in the same way as is described above.

Note that many of the commands that list constructs (`facts`, `list-deftemplates`, `rules`, etc) accept a module name or "*" as an optional argument. If no argument is specified, these commands operate only on the current module. If a module name is given, they operate on the named module. If "*" is given, they operate on all modules.

2.10.3. Module focus and execution control

In the previous sections I described how modules provide a kind of namespace facility, allowing you to partition a rulebase into manageable chunks. Modules can also be used to control execution. In general, although any Jess rule can be activated at any time, only rules in the *focus module* will fire. Note that the *focus module* is independent from the *current module* discussed above.

Initially, the module MAIN has the focus:

```
Jess> (defmodule DRIVING)
  TRUE
  Jess> (defrule get-in-car
    =>
    (printout t "Ready to go!" crlf))
  TRUE
  Jess> (reset)
  TRUE
  Jess> (run)
  0
```

In the example above, the rule doesn't fire because the DRIVING module doesn't have the focus. You can move the focus to another module using the `focus` function (which returns the name of the previous focus module:)

```
Jess> (focus DRIVING)
  MAIN
  Jess> (run)
    Ready to go!
    1
```

Note that you can call `focus` from the right-hand-side of a rule to change the focus while the engine is running.

Jess actually maintains a *focus stack* containing an arbitrary number of modules. The focus module is, by definition, the module on top of the stack. When there are no more activated rules in the focus module, it is "popped" from the stack, and the next module underneath becomes the focus module. You also can manipulate the focus stack with the functions `pop-focus, list-focus-stack, get-focus-stack,` and `clear-focus-stack.`

The example program `dilemma.clp` shows a good use of modules for execution control.

2.10.3.1. The auto-focus declaration

You can declare that a rule has the *auto-focus property*:

```
Jess> (defmodule PROBLEMS)
    TRUE
  Jess> (defrule crash
    (declare (auto-focus TRUE))
    (DRIVING::me ?location)
    (DRIVING::other-car ?location)
    =>
    (printout t "Crash!" crlf)
    (halt))
    TRUE
  Jess> (defrule DRIVING::travel
    ?me <- (me ?location)
    =>
    (printout t ".")
    (retract ?me)
    (assert (me (+ ?location 1))))
    TRUE
  Jess> (assert (me 1))
    <Fact-1>
  Jess> (assert (other-car 4))
    <Fact-2>
  Jess> (focus DRIVING)
    MAIN
  Jess> (run)
  ...Crash!
    4
```

When an auto-focus rule is activated, the module it appears in is automatically pushed onto the focus stack and becomes the focus module. Modules with auto-focus rules make great "background tasks."

2.10.3.2. Returning from a rule RHS

If the function `return` is called from a rule's right-hand-side, it immediately terminates the execution of that rule's RHS. Furthermore, the current focus module is popped from the focus stack.

This suggests that you can call a module like a subroutine. You call the module from a rule's RHS using `focus,` and you return from the call using `return.`

Back to index

3. Programming in the Jess Language

Useful expert systems can be written using the Jess language, with no extensions. I won't present a tutorial on writing such systems here (maybe someday!), but I do want to share a few useful hints and ideas in the following sections.

3.1. Using an External Editor

Jess allows you to enter rules and other code directly at its interactive prompt. While this is fine for experimenting, Jess doesn't yet have the ability to save the source text for all the rules and constructs you enter. Therefore, you will typically enter your rules and other data into a separate script file and read it into Jess using the `batch` command. Jess does offer the `ppdefrule` and `save-facts` commands, both of which can be very helpful in interactively building up a system definition and then storing it in a file. Note that you might use the `system` command to start the external editor from within Jess, if desired.

3.2. Efficiency of rule-based systems

The single biggest determinant of Jess performance is the number of *partial matches* generated by your rules. You should always try to obey the following (sometimes contradictory) guidelines while writing your rules:

- Put the *most specific* patterns (those that will match the fewest facts) near the top of each rule's LHS.
- Put the *most transient* patterns (those that will match facts that are frequently retracted and asserted) near the bottom of a LHS.

You can use the `view` command to find out how many partial matches your rules generate. See this chapter on How Jess Works for more details.

3.3. Error Reporting and Debugging

I'm constantly trying to improve Jess's error reporting, but it is still not perfect. When you get an error from Jess (during parsing or at runtime) it is generally delivered as a Java exception. The exception will contain an explanation of the problem and the stack trace of the exception will help you understand what went wrong. For this reason, it is *very important* that, if you're embedding Jess in a Java application, you don't write code like this:

```
try
{
  Rete engine = new Rete();
  engine.executeCommand("(gibberish!)");
}
catch (JessException re) { /* ignore errors */ }
```

If you ignore the Java exceptions, you will miss Jess's explanations of what's wrong with your code. Don't laugh - more people code this way than you'd think!

Anyway, as an example, if you attempt to load the folowing rule in the standard Jess command-line executable,

```
Jess> (defrule foo-1
        (foo bar)
        ->
        (printout "Found Foo Bar" crlf))
```

You'll get the following printout:

```
  Jess reported an error in routine Jesp.parseDefrule.
    Message: Expected '=>' .
    Program text: ( defrule foo-1 ( foo bar ) ->  at line 2.
          at jess.Jesp.parseError(Jesp.java:1434)
          at jess.Jesp.doParseDefrule(Compiled Code)
          at jess.Jesp.parseDefrule(Jesp.java:882)
          at jess.Jesp.parseSexp(Jesp.java:153)
          at jess.Jesp.parse(Compiled Code)
          at jess.Main.execute(Compiled Code)
          at jess.Main.main(Main.java:26)
```

This exception, like all exceptions reporte by Jess, lists a Java routine name. The name `parseDefrule` makes it fairly clear that a rule was being parsed, and the detail message explains that `->` was found in the input instead of the expected `=>` symbol (we accidentally typed `->` instead). This particular error message, then, was fairly easy to understand.

Runtime errors can be more puzzling, but the printout will generally give you a lot of information. Here's a rule where we erroneously try to add the number `3.0` to the word `four`:

```
Jess> (defrule foo-2
        =>
      (printout t (+ 3.0 four) crlf))
```

This rule will compile fine, since the parser doesn't know that the `+` function won't accept the atom `four` as an argument. When we `(reset)` and `(run)`, however, we'll see:

```
Jess reported an error in routine Value.numericValue
  while executing (+ 3.0 four) while executing (printout t (+ 3.0 four) crlf)
  while executing defrule foo-2 while executing (run).
  Message: Not a number: "four" (type = ATOM).
  Program text: ( run )  at line 4.
        at jess.Value.typeError(Value.java:361)
        at jess.Value.typeError(Value.java:356)
        at jess.Value.numericValue(Value.java:244)
        at jess.Plus.call(Compiled Code)
        at jess.FunctionHolder.call(FunctionHolder.java:35)
        at jess.Funcall.execute(Funcall.java:238)
        at jess.FuncallValue.resolveValue(FuncallValue.java:33)
        at jess.Printout.call(Compiled Code)
        at jess.FunctionHolder.call(FunctionHolder.java:35)
        at jess.Funcall.execute(Funcall.java:238)
        at jess.Defrule.fire(Compiled Code)
        at jess.Activation.fire(Activation.java:58)
        at jess.Rete.run(Compiled Code)
        at jess.Rete.run(Compiled Code)
        at jess.HaltEtc.call(Funcall.java:1559)
        at jess.FunctionHolder.call(FunctionHolder.java:35)
        at jess.Funcall.execute(Funcall.java:238)
```

```
        at jess.Jesp.parseAndExecuteFuncall(Jesp.java:1423)
        at jess.Jesp.parseSexp(Jesp.java:172)
        at jess.Jesp.parse(Compiled Code)
        at jess.Main.execute(Compiled Code)
        at jess.Main.main(Main.java:26)
```

In this case, the error message is also pretty clear. It shows the offending function (+ 3.0 four) then the function that called that (printout) then the context in which the function was called (defrule foo-2), and finally the function which caused the rule to fire (run).

Looking at the stack trace, starting from the top down, you can find entries for the + fucntion (`Plus.call()`), the `printout` function, the rule firing (`Defrule.fire()`) and the `run` command (`Rete.run()`).

The message 'Not a number: "four" (type = ATOM).' tells you that the + function wanted a numeric argument, but found the symbol (or ATOM) `four` instead.

If we make a similar mistake on the LHS of a rule:

```
        Jess> (defrule foo-3
                (test (eq 3 (+ 2 one)))
                 =>
                )
```

We see the following after a `reset`:

```
  Jess reported an error in routine Value.numericValue
    while executing (+ 2 one) while executing (eq 3 (+ 2 one))
    while executing 'test' CE while executing rule LHS (TECT) while executing (reset).
    Message: Not a number: "one" (type = ATOM).
    Program text: ( reset )  at line 4.
        at jess.Value.typeError(Value.java:361)
        at jess.Value.typeError(Value.java:356)
        at jess.Value.numericValue(Value.java:244)
        at jess.Plus.call(Compiled Code)
        at jess.FunctionHolder.call(FunctionHolder.java:35)
        at jess.Funcall.execute(Funcall.java:238)
        at jess.FuncallValue.resolveValue(FuncallValue.java:33)
        at jess.Eq.call(Compiled Code)
        at jess.FunctionHolder.call(FunctionHolder.java:35)
        at jess.Funcall.execute(Funcall.java:238)
        at jess.FuncallValue.resolveValue(FuncallValue.java:33)
        at jess.Test1.doTest(Test1.java:95)
        at jess.NodeJoin.runTests(Compiled Code)
        at jess.NodeJoin.callNode(Compiled Code)
        at jess.Node.passAlong(Compiled Code)
        at jess.Node1TECT.callNode(Compiled Code)
        at jess.Rete.processTokenOneNode(Rete.java:1009)
        at jess.Rete.processToken(Compiled Code)
        at jess.Rete.assert(Rete.java:754)
        at jess.Rete.reset(Rete.java:680)
        at jess.HaltEtc.call(Funcall.java:1565)
        at jess.FunctionHolder.call(FunctionHolder.java:35)
        at jess.Funcall.execute(Funcall.java:238)
        at jess.Jesp.parseAndExecuteFuncall(Jesp.java:1423)
```

```
        at jess.Jesp.parseSexp(Jesp.java:172)
        at jess.Jesp.parse(Compiled Code)
        at jess.Main.execute(Compiled Code)
        at jess.Main.main(Main.java:26)
```

Again, the error message is very detailed, and makes it clear, I hope, that the error occurred during rule LHS execution, in a `test` CE, in the function `(+ 2 one).` Note that Jess cannot tell you which rule this LHS belongs to, since rule LHSs can be shared.

3.4. Putting Java Objects into Fact Slots

You can easily put Java objects into the slots of Jess facts, as described elsewhere in this document. This section describes some minimal requirements for objects used in this way.

Jess uses the `equals` and `hashCode` methods of any objects added to a slot in a Jess fact. As such, it is very important that these methods be implemented properly. The Java API documentation lists some important properties of `equals` and `hashCode`, but I will reiterate the most important (and most often overlooked) one here: if you write `equals`, you probably *must* write `hashCode` too. For any two instance of a class for which `equals` returns `true`, `hashCode` must return the same value. If this rule is not observed, Jess will appear to malfunction when processing facts containing these improperly defined objects in their slots. In particular, rules that should fire may not do so.

Back to index

4. Introduction to Programming with Jess in Java

There are two main ways in which Java code can be used with Jess: Java can be used to extend Jess, and the Jess library can be used from Java. The material in this section is relevant to both of these endeavors. Refer to the API documentation for the complete story on these classes.

> *Note:* the code samples herein are necessarily not complete Java programs. In general, all excerpted code would need to appear inside a try block, inside a Java method, inside a Java class, to compile; and all Java source files are expected to include the `"import jess.*;"` declaration. Sometimes examples build on previous ones; this is usually clear from context. Such compound examples will need to be assembled into one method before compiling.

4.1. The jess.JessException class

The `jess.JessException` exception type is the only kind of exception thrown by any functions in the Jess library. `jess.JessException` is rather complex, as exception classes go. An instance of this class can contain a wealth of information about an error that occurred in Jess. Besides the typical error message, a `jess.JessException` may be able to tell you the name of the routine in which the error occurred, the name of the Jess constructs that were on the exceution stack, the relevant text and line number of the executing Jess language program, and the Java exception that triggered the error (if any.) See the the API documentation for details.

One of the most important pieces of advice for working with the Jess library is that in your catch clauses for JessException, *display the exception object*. Print it to System.out, or convert to a String and display it in a dialog box. The exceptions are there to help you by telling when something goes wrong; don't ignore them.

Another important tip: the JessException class has a method `getCause` which returns non-null when a particular JessException is a wrapper for another kind of exception. For example, if you use the Jess function `call` to call a function that throws an exception, then `call` will throw a JessException, and calling JessException.getCause() will return the real exception that was thrown. Your JessException handlers should always check getCause(); if your handler simply displays a thrown exception, then it should display the return value of getCause(), too. `getCause()` replaces the now deprecated `getNextException()`.

4.2. The jess.Value class

The class `jess.Value` is probably the one you'll use the most in working with Jess. A `Value` is a self-describing data object. Every datum in Jess is contained in one. Once it is constructed, a `Value`'s type and contents cannot be changed; it is *immutable*. `Value` supports a `type()` function, which returns one of these type constants (defined in the class `jess.RU` (RU = "Rete Utilities")):

```
final public static int NONE            =     0; ; an empty value (not NIL)
final public static int ATOM            =     1; ; a symbol
final public static int STRING          =     2; ; a string
final public static int INTEGER         =     4; ; an integer
final public static int VARIABLE        =     8; ; a variable
final public static int FACT            =    16; ; a jess.Fact object
```

```
final public static int FLOAT            =    32; ; a double float
final public static int FUNCALL          =    64; ; a function call
final public static int LIST             =   512; ; a multifield
final public static int DESCRIPTOR       =  1024; ; (internal use)
final public static int EXTERNAL_ADDRESS =  2048; ; a Java object
final public static int INTARRAY         =  4096; ; (internal use)
final public static int MULTIVARIABLE    =  8192; ; a multivariable
final public static int SLOT             = 16384; ; (internal use)
final public static int MULTISLOT        = 32768; ; (internal use)
final public static int LONG             = 65536; ; a Java long
```

Please always use the names, not the literal values, and the latter are subject to change without notice.

`Value` objects are constructed by specifying the data and (usually) the type. Each overloaded constructor assures that the given data and the given type are compatible. Note that for each constructor, more than one value of the `type` parameter may be acceptable. The available constructors are:

```
public Value(Object o) throws JessException
public Value(String s, int type) throws JessException
public Value(Value v)
public Value(ValueVector f, int type) throws JessException
public Value(double d, int type) throws JessException
public Value(int value, int type) throws JessException
```

`Value` supports a number of functions to get the actual data out of a `Value`object. These are

```
public Object externalAddressValue(Context c) throws JessException
public String stringValue(Context c) throws JessException
public Fact factValue(Context c) throws JessException
public Funcall funcallValue(Context c) throws JessException
public ValueVector listValue(Context c) throws JessException
public double floatValue(Context c) throws JessException
public double numericValue(Context c) throws JessException
public int intValue(Context c) throws JessException
```

The class `jess.Context` is described in the next section. If you try to convert random values by creating a Value and retrieving it as some other type, you'll generally get a JessException. However, some types can be freely interconverted: for example, integers and floats.

4.2.1. The subclasses of jess.Value

`jess.Value` has a number of subclasses: `jess.Variable`, `jess.FuncallValue`, `jess.FactIDValue`, and `jess.LongValue` are the four of most interest to the reader. When you wish to create a value to represent a variable, a function call, a fact, or a Java long, you must use the appropriate subclass.

> Note to the design-minded: I could have use a Factory pattern here and hidden the subclasses from the programmer. Because of the many different Value constructors, and for performance reasons, I decided this wouldn't be worth the overhead.

4.2.1.1. The class jess.Variable

Use this subclass of Value when you want to create a Value that represents a Variable. The one constructor looks like this:

```
public Variable(String s, int type) throws JessException
```

The type must be `RU.VARIABLE` or `RU.MULTIVARIABLE` or an exception will be thrown. The String argument is the name of the variable, without any leading '?' or '$' characters.

4.2.1.2. The class jess.FuncallValue

Use this subclass of Value when you want to create a Value that represents a function call (for example, when you are creating a `jess.Funcall` containing nested function calls.) The one constructor looks like this:

```
public FuncallValue(Funcall f) throws JessException
```

4.2.1.3. The class jess.LongValue

Use this subclass of Value when you want to create a Value that represents a Java long. These are mostly used to pass to Java functions called via reflection. The one constructor looks like

```
public LongValue(long l) throws JessException
```

4.2.1.4. The class jess.FactIDValue

Use this subclass of Value when you want to create a Value that represents a fact-id. The one constructor looks like this:

```
public FactIDValue(Fact f) throws JessException
```

In previous versions of Jess, fact-id's were more like integers; now they are really references to facts. As such, a fact-id must represent a valid jess.Fact object. Call `externalAddressValue(Context)` to get the jess.Fact object, and call `Fact.getFactId()` to get the fact-id as an integer. This latter manipulation will now rarely, if ever, be necessary.

4.2.2. Value resolution

Some `jess.Value` objects may need to be *resolved* before use. To resolve a `jess.Value` means to interpret it in a particular context. `jess.Value` objects can represent both static values (atoms, numbers, strings) and dynamic ones (variables, function calls). It is the dynamic ones that obviously have to be interpreted in context.

All the `jess.Value` member functions, like `intValue()`, that accept a `jess.Context` as an argument are *self-resolving;* that is, if a `jess.Value` object represents a function call, the call will be executed in the given `jess.Context`, and the `intValue()` method will be called on the result. Therefore, you often don't need to worry about resolution as it is done automatically. There are several cases where you will, however.

- *When interpreting arguments to a function written in Java.* The parameters passed to a Java Userfunction may themselves represent function calls. It may be important, therefore, that these values be resolved only once, as these functions may have side-effects (I'm tempted to use the computer-science word: these functions may not be *idempotent.* Idempotent functions have no side-effects and thus may be called multiple times without harm.) You can accomplish this by calling one of the `(x)Value()` methods and storing the return value, using this return value instead of the parameter itseld. Alternatively, you may call `resolveValue()`and store the return value in a new `jess.Value` variable, using this value as the new parameter. Note that the `type()` method will return `RU.VARIABLE` for a `jess.Value` object that refers to a variable, regardless of the type of the value the variable is bound to. The resolved value will return the proper type.

  Note that arguments to `deffunctions` are resolved automatically, before your Jess language code runs.
- *when returning a `jess.Value` object from a function written in Java.* If you return one of a function's parameters from a Java Userfunction, be sure to return the return value of `resolveValue()`, not the parameter itself.
- *When storing a `jess.Value` object.* It is important that any values passed out of a particular execution context be resolved; for example, before storing a Value object in a Hashtable, `resolveValue()` should always be called on both the key and object.

4.3. The jess.Context class

`jess.Context` represents an execution context for the evaluation of function calls and the resolution of variables. There are very few public member functions in this class, and only a few of general importance.

You can use `getVariable()` and `setvariable`to get and change the value of a variable from Java code, respectively.

The function `getEngine()` gives any `Userfunction` access to the Rete object in which it is executing.

When a `Userfunction` is called, a `jess.Context` argument is passed in as the final argument. You should pass this `jess.Context` to any `jess.Value.(x)Value()` calls that you make.

4.4. The jess.Rete class

The `jess.Rete` class is the rule engine itself. Each `jess.Rete` object has its own knowledge base, agenda, rules, etc. To embed Jess in a Java application, you'll simply need to create one or more `jess.Rete` objects and manipulate them appropriately. We'll cover this in more detail in the section on embedding Jess in Java applications. Here I will cover some general features of the `jess.Rete` class.

4.4.1. Equivalents for common Jess functions

Several of the most commonly used Jess functions are wrappers for methods in the `jess.Rete` class. Examples are `run()`, `run(int)`, `reset()`, `clear()`, `assertFact(Fact)`, `retract(Fact)`, `retract(int)`, and `halt()`. You can call these from Java just as you would from Jess.

4.4.2. Executing other Jess commands

You can use the `Rete` class's `executeCommand(String cmd)` method to easily execute, from Java, any Jess function call or construct definition that can be represented as a parseable String. For example,

```
import jess.*;
public class ExSquare
{
  public static void main(String[] unused)
  {
    try
      {
        Rete r = new Rete();
        r.executeCommand("(deffunction square (?n) (return (* ?n ?n)))");
        Value v = r.executeCommand("(square 3)");

        // Prints '9'
        System.out.println(v.intValue(r.getGlobalContext()));
      }
    catch (JessException ex)
      {
        System.err.println(ex);
      }
  }
}
C:\> java ExSquare
    9
```

`executeCommand()` returns the `jess.Value` object returned by the command. Commands executed via `executeCommand()` may refer to Jess variables; they will be interpreted in the *global context*. In general, only defglobals can be used in this way.

Note that you may only pass one function call or construct at a time to `executeCommand()`.

4.4.2.1. Optional commands

Note that when you create a Rete object from Java, it will already contain definitions for all of the functions that come with Jess. There are no longer any "optional" commands.

4.4.3. The script library

Some of Jess's commands are defined in Jess language code, in the file `jess/scriptlib.clp`. Each `Rete` object will load this script library when it is created and again if (clear) is called. In previous versions of Jess you had to do this yourself; this is no longer necessary.

4.4.4. Transferring values between Jess and Java code

This section describes a very easy-to-use mechanism for communicating inputs and results between Jess and Java code.

These methods are available in the class `jess.Rete:`

```
public Value store(String name, Value val);
public Value store(String name, Object val);
public Value fetch(String name);
public void clearStorage();
```

while these functions are available in Jess:

```
(store <name> <value>)
(fetch <name>)
(clear-storage)
```

Both `store` methods accept a "name" and a value (in Java, either in the form of a `jess.Value` object or an ordinary Java object; in Jess, any value), returning any old value with that name, or null (or nil in Jess) if there is none. Both `fetch` methods accept a name, and return any value stored under that name, or null/nil if there is no such object. These functions therefore let you transfer data between Jess and Java that cannot be represented textually (of course they work for Strings, too.) In this example we create an object in Java, then pass it to Jess to be used as an argument to the `definstance` command.

```
import jess.*;
public class ExFetch
{
  public static void main(String[] unused) throws JessException
  {
    Rete r = new Rete();
    r.store("DIMENSION", new java.awt.Dimension(10, 10));
    r.executeCommand("(defclass dimension java.awt.Dimension)");
    r.executeCommand("(definstance dimension (fetch DIMENSION) static)");

    r.executeCommand("(facts)");
  }
}
C:\> java ExFetch
    f-0   (MAIN::dimension (class <External-Address:java.lang.Class>) (height 10.0) (size <External-Address:java.awt.Dimension>) (width 10.0) (OBJECT <External-Address:java.awt.Dimension>))
    For a total of 1 facts.
```

Note that storing a null (or nil) value will result in the given name being removed from the hashtable altogether. `clearStorage()` and `clear-storage` each remove all data from the hashtable.

Note that the Jess `clear` and Java `clear()` functions will call `clearStorage(),` but `reset` and `reset()` will not. Stored data is thus available across calls to `reset()`.

Another example, with a main program in Java and a set of rules that return a result using `store` is in the directory Jess61RC1/jess/examples/xfer/ .

4.4.5. Methods for adding, finding and listing constructs

The easiest (and still encouraged) way to define templates, `defglobals,` and other constructs is to use Jess language code and let Jess parse the textual definition. However, many of these constructs are represented by public classes in the Jess library, and if you wish, you can construct your own instances of these in Java code and add them to an engine explicitly. This is currently possible for most, but not all, Jess constructs. Right now the `jess.Defrule` class does not expose enough public methods to properly create one outside of the `jess` package. This is deliberate, as this API is likely to change again in the near future. For information about the classes mentioned here (`jess.Deftemplate`, `jess.Defglobal`, etc) see the API documentation.

These `jess.Rete` methods let you add constructs to the engine:

- `public void addDeffacts(Deffacts)`
- `public void addDefglobal(Defglobal)`
- `public void addDefrule(Defrule)`
- `public void addDeftemplate(Deftemplate)`
- `public void addUserfunction(Userfunction)`
- `public void addUserpackage(Userpackage)`

These methods return individual constructs from within the engine, generally by name:

- `public Defglobal findDefglobal(String)`
- `public Defrule findDefrule(String)`
- `public Deftemplate findDeftemplate(String)`
- `public Userfunction findUserfunction(String)`

These methods return `java.util.Iterator`s of various data structures in the engine:

- `public Iterator listActivations()`
- `public Iterator listDeffacts()`
- `public Iterator listDefglobals()`
- `public Iterator listDefrules()`
- `public Iterator listDeftemplates()`
- `public Iterator listFacts()`
- `public Iterator listFunctions()`

4.4.6. I/O Routers

The functions `printout` and `format` take an *I/O router name* as an argument. The default argument `t` is Jess' `standard input and output`. Jess also has a special WSTDOUT router for printing user messages internally - for example, the Jess> prompt, the messages you get when you issue a `watch` command, and the output of commands like `facts` and `ppdefrule.` The `read` command and `readline` take input from the `t` router by default.

By default, Jess's standard routers are connected to Java's standard streams, so that output goes to the command-line window. This is perfect for command-line programs, but of course not acceptable for GUI-based applications. To remedy this, Jess lets you connect the `t` router to any Java `java.io.Reader` and `java.io.Writer` objects you choose. In fact, you can not only redirect the `t` router, but you can add routers of your own, in much the same way that the `open` command creates a new router that reads from a file.

These functions in the Rete class let you manipulate the router list:

- `public void addInputRouter(String s, Reader is, boolean consoleLike)`
- `public void addOutputRouter(String s, Writer os)`
- `public Reader getInputMode(String s)`
- `public Reader getInputRouter(String s)`

- `public Writer getOutputRouter(String s)`
- `public void removeInputRouter(String s)`
- `public void removeOutputRouter(String s)`

The words "input" and "output" are from the perspective of the Jess library itself; i.e., Jess reads from input routers and writes to output routers.

Note that you can use the same name for an input router and an output router (the `t` router is like that.) Note also that although these functions accept and return generic `Reader` and `Writer` objects, Jess internally uses `java.io.PrintWriter` and `java.io.BufferedReader`. If you pass in other types, Jess will construct one of these preferred classes to "wrap" the object you pass in.

When Jess starts up, there are three output routers and one input router defined: the `t` router, which reads and writes from the standard input and output; the `WSTDOUT` router, which Jess uses for all prompts, diagnostic outputs, and other displays; and the `WSTDERR` router, which Jess uses to print stack traces and error messages. By default, `t` is connected to `System.in` and `System.out`, and both `WSTDOUT` and `WSTDERR` are connected to `System.out` (neither is connected to `System.err`.) You can reroute these inputs and outputs simply by changing the Readers and Writers they are attached to using the above functions. You can use any kind of streams you can dream up: network streams, file streams, etc.

The `boolean` argument `consoleLike` to the `addInputRouter` method specifies whether the stream should be treated like the standard input or like a file. The difference is that on console-like streams, a `read` call consumes an entire line of input, but only the first token is returned; while on file-like streams, only the characters that make up each token are consumed on any one call. That means, for instance, that a `read` followed by a `readline` will consume two lines of text from a console-like stream, but only one from a file-like stream, given that the first line is of non-zero length. This odd behaviour is actually just following the behaviour of CLIPS.

The `jess.Rete` class has two more handy router-related methods: `getOutStream()` and `getErrStream()`, both of which return a `java.io.PrintWriter` object. `getOutStream()` returns a stream that goes to the same place as the current setting of `WSTDOUT`; `errStream()` does the same for `WSTDERR`.

4.4.7. jess.awt.TextAreaWriter and jess.awt.TextReader

Jess ships with two utility classes that can be very useful when building GUIs for Jess: the `jess.awt.TextAreaWriter` and `jess.awt.TextReader` classes. Both can serve as adapters between Jess and graphical input/output widgets. The `TextAreaWriter` class is, as the name implies, a Java `Writer` that sends any data written to it to a `java.awt.TextArea`. This lets you place Jess's output in a scrolling window on your GUI. The `jess.Console` and `jess.ConsoleApplet` jess GUIs use these classes. To use `TextAreaWriter`, simply call `addOutputRouter()`, passing in an instance of this class:

```
import java.awt.TextArea;
import jess.awt.*;
import jess.*;
public class ExTAW
```

```
  {
    public static void main(String[] unused) throws JessException
    {
       TextArea ta = new TextArea(20, 80);
       TextAreaWriter taw = new TextAreaWriter(ta);

       Rete r = new Rete();
       r.addOutputRouter("t", taw);
       r.addOutputRouter("WSTDOUT", taw);
       r.addOutputRouter("WSTDERR", taw);
       // Do something interesting, then...
       System.exit(0);
    }
  }
C:\> java ExTAW
```

Now the output of the `printout` command, for example, will go into a scrolling window (of course, you need to display the `TextArea` on the screen somehow!) Study `jess/ConsolePanel.java` and `jess/Console.java` to see a complete example of this.

`jess.awt.TextReader` is similar, but it is a `Reader` instead. It is actually quite similar to `java.io.StringReader`, except that you can continually add new text to the end of the stream (using the `appendText()` method). It is intended that you create a `jess.awt.TextReader`, install it as an input router, and then (in an AWT event handler, somewhere) append new input to the stream whenever it becomes available. See the same `jess/Console*` files for a complete usage example for this class as well.

4.5. The jess.ValueVector class

The `jess.ValueVector` class is Jess's internal representation of a *list*, and therefore has a central role in programming with Jess in Java. The `jess.ValueVector` class itself is used to represent generic lists (multifields), while specialized subclasses are used as function calls (`jess.Funcall`), facts (`jess.Fact`), and templates (`Deftemplate`).

Working with `ValueVector` itself is simple. Its API is reminiscent of `java.util.Vector`. Like that class, it is a self-extending array: when new elements are added the `ValueVector` grows in size to accomodate them. Here is a bit of example Java code in which we create the Jess list `(a b c)`.

```
import jess.*;
public class ExABC
{
  public static void main(String[] unused) throws JessException
  {
    ValueVector vv = new ValueVector();
    vv.add(new Value("a", RU.ATOM));
    vv.add(new Value("b", RU.ATOM));
    vv.add(new Value("c", RU.ATOM));

    // Prints "(a b c)"
    System.out.println(vv.toStringWithParens());
  }
```

```
}
C:\> java ExABC
    (a b c)
```

The `add()` function returns the `ValueVector` object itself, so that `add()` calls can be chained together for convenience:

```
import jess.*;
public class ExChain
{
  public static void main(String[] unused) throws JessException
  {
    ValueVector vv = new ValueVector();
    vv.add(new Value("a", RU.ATOM)).add(new Value("b", RU.ATOM)).add(new Value("c", RU.ATOM));

    // Prints "(a b c)"
    System.out.println(vv.toStringWithParens());
  }
}
C:\> java ExChain
    (a b c)
```

To pass a list from Java to Jess, you should enclose it in a `jess.Value` object of type `RU.LIST`.

4.6. The jess.Funcall class

`jess.Funcall` is a specialized subclass of `ValueVector` that represents a Jess function call. It contains the name of the function, an internal pointer to the actual `jess.Userfunction` object containing the function code, and the arguments to pass to the function.

You can call Jess functions using `jess.Funcall` if you prefer, rather than using `jess.Rete.executeFunction()`. This method has less overhead since there is no parsing to be done. This example is an alternate version of the "defclass Dimension" example above.

```
import java.awt.Dimension;
import jess.*;
public class ExADimension
{
  public static void main(String[] unused) throws JessException
  {
    Rete r = new Rete();
    Context c = r.getGlobalContext();
    Value dimension = new Value("dimension", RU.ATOM);

    Funcall f = new Funcall("defclass", r);
    f.arg(dimension).arg(new Value("java.awt.Dimension", RU.ATOM));
    f.execute(c);

    new Funcall("definstance", r).arg(dimension).
        arg(new Value(new Dimension(10, 10))).
        arg(new Value("static", RU.ATOM)).execute(c);

    new Funcall("facts", r).execute(c);
  }
}
C:\> java ExADimension
    f-0   (MAIN::dimension (class <External-Address:java.lang.Class>) (height 10.0) (size <External-Address:java.awt.Dimension>) (width 10.0) (OBJECT <External-Address:java.awt.Dimension>))
    For a total of 1 facts.
```

The example shows several styles of using `jess.Funcall`. You can chain `add()` calls, but remember that `add()` returns `ValueVector`, so you can't call `execute()` on the return value of `Funcall.add()` A special method `arg()` is provided for this purpose; it does the same thing as `add()` but returns the `Funcall` as a `Funcall`.

The first entry in a `Funcall`'s `ValueVector` is the name of the function, even though you don't explicitly set it. Changing the first entry will not automatically change the function the `Funcall` will call!

The `Funcall` class also contains some public static constant `Value` member objects that represent the special atoms `nil, TRUE, FALSE, EOF,` etc. You are encouraged to use these.

4.7. The jess.Fact class

Another interesting subclass of `ValueVector` is `jess.Fact,` which, predictably, is how Jess represents facts. A `Fact` is stored as a list in which all the entries correspond to slots. The head or name of the fact is stored in a separate variable (available via the `getName()` method.)

Once you assert a jess.Fact object, you no longer "own" it - it becomes part of the Rete object's internal data structures. As such, you must not change the values of any of the Fact's slots. If you retract the fact, the Fact object is released and you are free to alter it as you wish. Alternatively, you can use the jess.Rete.modify() method to modify a fact.

4.7.1. Constructing an Unordered Fact from Java

In the following example, we create a template and assert an unordered fact that uses it.

```
import jess.*;
public class ExPoint
{
  public static void main(String[] unused) throws JessException
  {
    Rete r = new Rete();
    r.executeCommand("(deftemplate point \"A 2D point\" (slot x) (slot y))");

    Fact f = new Fact("point", r);
    f.setSlotValue("x", new Value(37, RU.INTEGER));
    f.setSlotValue("y", new Value(49, RU.INTEGER));
    r.assertFact(f);

    r.executeCommand("(facts)");
  }
}
C:\> java ExPoint
    f-0   (MAIN::point (x 37) (y 49))
  For a total of 1 facts.
```

4.7.2. Constructing a Multislot from Java

In this example, the template has a multislot. In Java, a multislot is represented by a `Value` of type `RU.LIST`; the `Value` object contains a `ValueVector` containing the fields of the multislot.

```
import jess.*;
public class ExMulti
{
  public static void main(String[] unused) throws JessException
  {
    Rete r = new Rete();
    r.executeCommand("(deftemplate vector \"A named vector\"" +
                     " (slot name) (multislot list))");

    Fact f = new Fact("vector", r);
    f.setSlotValue("name", new Value("Groceries", RU.ATOM));
    ValueVector vv = new ValueVector();
    vv.add(new Value("String Beans", RU.STRING));
    vv.add(new Value("Milk", RU.STRING));
    vv.add(new Value("Bread", RU.STRING));
    f.setSlotValue("list", new Value(vv, RU.LIST));
    r.assertFact(f);

    r.executeCommand("(facts)");
  }
}
C:\> java ExMulti
    f-0   (MAIN::vector (name Groceries) (list "String Beans" "Milk" "Bread"))
  For a total of 1 facts.
```

4.7.3. Constructing an Ordered Fact from Java

An ordered fact is actually represented as an unordered fact with a single slot: a multislot named __data. You don’t need to create a template for an ordered fact: one will be created automatically if it doesn’t already exist.

```
import jess.*;
public class ExOrdered
{
  public static void main(String[] unused) throws JessException
  {
    Rete r = new Rete();

    Fact f = new Fact("letters", r);
    ValueVector vv = new ValueVector();
    vv.add(new Value("a", RU.ATOM));
    vv.add(new Value("b", RU.ATOM));
    vv.add(new Value("c", RU.ATOM));
    f.setSlotValue("__data", new Value(vv, RU.LIST));
    r.assertFact(f);

    r.executeCommand("(facts)");
  }
}
```

```
C:\> java ExOrdered
    f-0   (MAIN::letters a b c)
  For a total of 1 facts.
```

4.8. The jess.Deftemplate class

Yet another interesting subclass of `ValueVector` is `jess.Deftemplate,` the purpose of which should be obvious. `Deftemplate` has a fairly large interface which allows you to set and query the properties of a template's slots.

This example is an alternative to the `deftemplate` command in the previous example.

```
import jess.*;
public class ExBuildDeftemplate
{
  public static void main(String[] unused) throws JessException
  {
    Rete r = new Rete();
    Deftemplate dt = new Deftemplate("point", "A 2D point", r);
    Value zero = new Value(0, RU.INTEGER);
    dt.addSlot("x", zero, "NUMBER");
    dt.addSlot("y", zero, "NUMBER");
    r.addDeftemplate(dt);

    // Now create and assert Fact
  }
}
```

4.9. The jess.Token class

The `jess.Token` class is used to represent partial matches in the Rete network. You'll use it if you're writing an Accelerator (not documented here) or if you're working with queries.

Only a few methods of `jess.Token` are public, and fewer are of use to the programmer. `int size()` tells you how many `jess.Fact`s are in a given `jess.Token.` The most important method is `Fact fact(int)`, which returns the `jess.Fact` objects that make up the partial match. Its argument is the zero-based index of the `jess.Fact` to retrieve, and must be between 0 and the return value of `size().` Each `Fact` will correspond to one pattern on a rule or query LHS; dummy facts are inserted for `not` and `test` CEs.

4.10. The jess.JessEvent and jess.JessListener classes

`jess.JessEvent` and `jess.JessListener` make up Jess's rendition of the standard Java event pattern. By implementing the `JessListener` interface, a class can register itself with a source of `JessEvent`s, like the `jess.Rete` class. `jess.Rete` (potentially) fires events at all critical junctures during its execution: when rules fire, when a `reset()` or `clear()` call is made, when a fact is asserted or retracted, etc. `JessEvent` has a `getType()` method to tell you what sort of event you have been notified of; the type will be one of the constants in the `JessEvent` class.

You can control which events a `jess.Rete` object will fire using the `setEventMask()` method. The argument is the result of logical-OR-ing together some of the constants in the `jess.JessEvent` class. By default, the event mask is 0 and no events are sent.

As an example, let's suppose you'd like your program's graphical interface to display a running count of the number of facts on the fact-list, and the name of the last executed rule. You've provided a static method, `MyGUI.displayCurrentRule(String ruleName),` which you would like to have called when a rule fires. You've got a pair of methods `MyGUI.incrementFactCount()` and `MyGUI.decrementFactCount()` to keep track of facts. And you've got one more static method, `MyGUI.clearDisplay()`, to call when Jess is cleared or reset. To accomplish this, you simply need to write an event handler, install it, and set the event mask properly. Your event handler class might look like this.

```
import jess.*;

public class ExMyEventHandler implements JessListener
{
  public void eventHappened(JessEvent je)
  {
    int defaultMask = JessEvent.DEFRULE_FIRED | JessEvent.FACT |
                      JessEvent.RESET | JessEvent.CLEAR;
    int type = je.getType();
    switch (type)
    {
      case JessEvent.CLEAR:
        Rete engine = (Rete) je.getSource();
        int mask = engine.getEventMask();
        mask |= defaultMask;
        engine.setEventMask(mask);

      // FALL THROUGH
      case JessEvent.RESET:
        // MyGUI.clearDisplay();
        break;

      case JessEvent.DEFRULE_FIRED:
        // MyGUI.displayCurrentRule( ((Activation) je.getObject()).getRule().getName());
        break;

      case JessEvent.FACT | JessEvent.REMOVED:
        // MyGUI.decrementFactCount();
        break;

      case JessEvent.FACT:
        // MyGUI.incrementFactCount();
        break;

      default:
        // ignore
    }
  }
}
```

Note how the event type constant for fact retracting is composed from `FACT | REMOVED.` In general,

constants like DEFRULE, DEFTEMPLATE, etc, refer to the addition of a new construct, while composing these with REMOVE signifies the removal of the same construct.

The getObject() method returns ancillary data about the event. In general, it is an instance of the type of object the event refers to; for DEFRULE_FIRED it is a jess.Defrule.

To install this listener, you would simply create an instance and call jess.Rete.addEventListener(), then set the event mask:

```
  import jess.*;
  public class ExMask
  {
    public static void main(String[] unused) throws JessException
    {
      Rete engine = new Rete();
      engine.addJessListener(new ExMyEventHandler());
      engine.setEventMask(engine.getEventMask() |
                          JessEvent.DEFRULE_FIRED | JessEvent.CLEAR |
                          JessEvent.FACT | JessEvent.RESET );
    }
  }
C:\> java ExMask
```

One added wrinkle: note how the handler for JessEvent.CLEAR sets the event mask. When (clear) is called, the event mask is typically reset to the default. When event handlers are called, they have the opportunity to alter the mask to re-install themselves. Alternatively, they can call removeJessListener() to unregister themselves.

Note that each event handler added will have a negative impact on Jess performance so their use should be limited.

There is no way to receive only one of an event / (event | REMOVE) pair.

4.10.1. Working with events from the Jess language

It's possible to work with the event classes from Jess language code as well. To write an event listener, you can use the jess.JessEventAdapter class. This class works rather like the jess.awt adapter classes do. Usage is best illustrated with an example. Let's say you want to print a message each time a new template is defined, and you want to do it from Jess code. Here it is:

```
Jess> ;; make code briefer
  (import jess.*)
  TRUE

  Jess> ;; Here is the event-handling deffunction
  ;; It accepts one argument, a JessEvent
  (deffunction display-deftemplate-from-event (?evt)
    (if (eq (get-member JessEvent DEFTEMPLATE) (get ?evt type)) then
      (printout t "New deftemplate: " (call (call ?evt getObject) getName) crlf)))
  TRUE
```

```
Jess> ;; Here we install the above function using a JessEventAdapter
(call (engine) addJessListener
(new JessEventAdapter display-deftemplate-from-event (engine)))

Jess> ;; Now we add DEFTEMPLATE to the event mask
(set (engine) eventMask
(bit-or (get (engine) eventMask) (get-member JessEvent DEFTEMPLATE)))
```

Now whenever a new template is defined, a message will be displayed.

4.11. Setting and Reading Java Bean Properties

As mentioned previously, Java objects can be explicitly pattern-matched on the LHS of rules, but only to the extent that they are *Java Beans*. A Java Bean is really just a Java object that has a number of methods that obey a simple naming convention for *Java Bean properties.* A class has a Bean property if, for some string *X* and type *T* it has either or both of:

- A method named `get`*X* which returns *T* and accepts no arguments; or, if *T* is boolean, named `is`*X* which accepts no arguments;
- A method named `set`X which returns void and accepts a single argument of type *T*.

Note that the capitalization is also important: for example, for a method named isVisible, the property's name is *visible*, with a lower-case V. Only the capitalization of the first letter of the name is important. You can conveniently set and get these properties using the Jess `set` and `get` methods. Note that many of the trivial changes in the Java 1.1 were directed towards making most visible properties of objects into Bean properties.

An example: AWT components have many Bean properties. One is *visible*, the property of being visible on the screen. We can read this property in two ways: either by explicitly calling the `isVisible()` method, or by querying the Bean property using `get.`

```
Jess> (defglobal ?*frame* = (new java.awt.Frame "Frame Demo"))
  TRUE

  Jess> ;; Directly call 'isVisible', or...
  (printout t (call ?*frame* isVisible) crlf)
  FALSE

  Jess> ;; ... equivalently, query the Bean property
  (printout t (get ?*frame* visible) crlf)
  FALSE
```

4.12. Formatting Jess Constructs

The class `jess.PrettyPrinter` can produce a formatted rendering of many Jess objects, including `jess.Defrule`s, `Deffunction`s `jess.Defquery`s, etc -- anything that implements the `jess.Visitable` interface. `jess.PrettyPrinter` is very simple to use: you just create an instance, passing the object to be rendered as a constructor argument, and then call `toString` to get the formatted result.

```
import jess.*;
public class ExPretty
{
  public static void main(String[] unused) throws JessException
  {
    Rete r = new Rete();
    r.executeCommand("(defrule myrule (A) => (printout t \"A\" crlf))");
    Defrule dr = (Defrule) r.findDefrule("myrule");
    System.out.println(new PrettyPrinter(dr));
  }
}
C:\> java ExPretty
    (defrule MAIN::myrule
    (MAIN::A)
    =>
    (printout t "A" crlf))
```

Back to index

5. Adding Commands to Jess

The Java interface `jess.Userfunction` represents a single function in the Jess language. You can add new functions to the Jess language simply by writing a class that implements the `jess.Userfunction` interface (see below for details on how this is done), creating a single instance of this class and installing it into a `jess.Rete` object using `Rete.addUserfunction()`. The `Userfunction` classes you write can maintain state; therefore a `Userfunction` can cache results across invocations, maintain complex data structures, or keep references to external Java objects for callbacks. A single `Userfunction` can be a gateway into a complex Java subsystem.

5.1. Writing Extensions

I've made it as easy as possible to add user-defined functions to Jess. There is no system type-checking on arguments, so you don't need to tell the system about your arguments, and values are self-describing, so you don't need to tell the system what type you return. You do, however, need to understand several Jess classes, including `jess.Value`, `jess.Context`, and `jess.Funcall`, as previously discussed in the chapter Introduction to Programming with Jess in Java.

5.1.1. Implementing your Userfunction

To implement the `jess.Userfunction` interface, you need to implement only two methods: `getName()` and `call()`. Here's an example of a class called 'MyUpcase' that implements the Jess function `my-upcase`, which expects a String as an argument, and returns the string in uppercase.

```
import jess.*;

public class ExMyUpcase implements Userfunction
  {
    // The name method returns the name by which the function will appear in Jess code.
    public String getName() { return "my-upcase"; }

    public Value call(ValueVector vv, Context context) throws JessException
      {
        return new Value(vv.get(1).stringValue(context).toUpperCase(), RU.STRING);
      }
  }
```

The `call()` method does the business of your Userfunction. When `call()` is invoked, the first argument will be a `ValueVector` representation of the Jess code that evoked your function. For example, if the following Jess function calls were made,

```
Jess> (load-function ExMyUpcase)
   Jess> (my-upcase foo)
   "FOO"
```

the first argument to `call()` would be a `ValueVector` of length two. The first element would be a `Value` containing the symbol (type `RU.ATOM`) `my-upcase`, and the second argument would be a `Value` containing the string (`RU.STRING`) `"foo"`.

Note that we use `vv.get(1).stringValue(context)` to get the first argument to `my-upcase` as a Java String. If the argument doesn't contain a string, or something convertible to a string, `stringValue()` will throw a `JessException` describing the problem; hence you don't need to worry about incorrect argument types if you don't want to. `vv.get(0)` will always return the symbol `my-upcase`, the name of the function being called (the clever programmer will note that this would let you construct multiple objects of the same class, implementing different functions based on the name of the function passed in as a constructor argument). If you want, you can check how many arguments your function was called with and throw a JessException if it was the wrong number by using the `vv.size()` method. In any case, our simple implementation extracts a single argument and uses the Java `toUpperCase()` method to do its work. `call()` must wrap its return value in a `jess.Value` object, specifying the type (here it is `RU.STRING`).

5.1.1.1. Legal return values

A `Userfunction` must return a valid `jess.Value` object; it cannot return tghe Java `null` value. To return "no value" to Jess, use `nil`. The value of `nil` is available in the public static final variable `jess.Funcall.NIL.`

5.1.2. Loading your Userfunction

Having written this class, you can then, in your Java main program, simply call `Rete.addUserfunction()` with an instance of your new class as an argument, and the function will be available from Jess code. So, we could have

```
import jess.*;
public class ExAddUF
{
  public static void main(String[] argv) throws JessException
  {
    // Add the 'my-upcase' command to Jess
    Rete r = new Rete();
    r.addUserfunction(new ExMyUpcase());

    // This will print "FOO".
    r.executeCommand("(printout t (my-upcase foo) crlf)");
  }
}
C:\> java ExAddUF
    FOO
```

Alternatively, the Jess language command `load-function` could be used to load `my-upcase` from Jess:

```
Jess> (load-function ExMyUpcase)
  Jess> (printout t (my-upcase foo) crlf)
  FOO
```

5.1.3. Calling assert from a Userfunction

The `jess.Rete.assertFact()` method has two overloads: one version takes a `jess.Context` argument, and the other does not. When writing a Userfunction, you should always use the first version, passing the `jess.Context` argument to the Userfunction. If you do not, your Userfunction will not interact correctly with the `logical` conditional element.

5.2. Writing Extension Packages

The `jess.Userpackage` interface is a handy way to group a collection of `Userfunctions` together, so that you don't need to install them one by one (all of the extensions shipped with Jess are included in Userpackage classes). A `Userpackage` class should supply the one method `add(),` which should simply add a collection of `Userfunctions` to a `Rete` object using `addUserfunction().` Nothing mysterious going on, but it's very convenient. As an example, suppose `MyUpcase` was only one of a number of similar functions you wrote. You could put them in a `Userpackage` class like this:

```
import jess.*;
public class ExMyStringFunctions implements Userpackage
  {
    public void add(Rete engine)
      {
        engine.addUserfunction(new ExMyUpcase());
        // Other similar statements
      }
  }
```

Now in your Java code, you can call

```
import jess.*;
public class ExAddUP
{
  public static void main(String[] argv) throws JessException
  {
    // Add the 'my-upcase' command to Jess
    Rete r = new Rete();
    r.addUserpackage(new ExMyStringFunctions());

    // This will still print FOO.
    r.executeCommand("(printout t (my-upcase foo) crlf)");
  }
}
```

```
C:\> java ExAddUP
FOO
```

or from your Jess code, you can call

```
Jess> (load-package ExMyStringFunctions)
Jess> (printout t (my-upcase foo) crlf)
FOO
```

to load these functions in. After either of these snippets, Jess language code could call `my-upcase`, `my-downcase`, etc.

`Userpackage`s are a great place to assemble a collection of interrelated functions which potentially can share data or maintain references to other function objects. You can also use `Userpackage`s to make sure that your `Userfunction`s are constructed with the correct constructor arguments.

All of Jess's "built-in" functions are simply `Userfunctions`, albeit ones which have special access to Jess' innards. Most of them are automatically loaded by code in the `jess.Funcall` class. You can use these as examples for writing your own Jess extensions.

5.3. Obtaining References to Userfunction Objects

Occasionally it is useful to be able to obtain a reference to an installed `Userfunction` object. The method `Userfunction Rete.findUserfunction(String name)` lets you do this easily. It returns the `Userfunction` object registered under the given name, or null if there is none. This is useful when you write Userfunctions which themselves maintain state of some kind, and you need access to that state.

Back to index

6. Embedding Jess in a Java Application

6.1. Using the class jess.Main

The class `jess.Main` provides not only the Jess command-line interface, but also the backbone of the example graphical interfaces (`jess.Console` and `jess.ConsoleApplet`) as well. You can reuse `jess.Main` from your own applications, or (if you have a source distribution) you can simply use it as an example of what an application embedding Jess can look like. `jess.Main` does a number of things that a reasonable Jess application might do:

- It (optionally) reads an entire file of Jess code directly, using the `jess.Jesp` parser class.
- It reads and executes user input in a loop.

6.2. Manipulating Jess in other ways

Note that each individual `jess.Rete` object represents an independent reasoning engine. A single program can then include several independent engines.

Jess can be used in a multithreaded environment. The `jess.Rete` class internally synchronizes itself using several synchronization locks. The most important lock is a global lock on all rule LHSs: only one assert or retract may be processing in a given `jess.Rete` object at a time. This restriction is likely to be relaxed in the future.

Back to index

7. Creating Graphical User Interfaces in the Jess Language

Jess, being just a set of Java classes, is easily incorporated as a library into graphical applications written in Java. It is also possible, though, to write graphical applications in the Jess language itself. The details of this are outlined in this chapter.

7.1. Handling Java AWT events

It should now be obvious that you can easily construct GUI objects from Jess. For example, here is a `Button`:

```
Jess> (defglobal ?*b* = (new java.awt.Button "Hello"))
```

What should not be obvious is how, from Jess, you can arrange to have something happen when the button is pressed. For this, I have provided a full set of `EventListener` classes:

- `jess.awt.ActionListener`
- `jess.awt.AdjustmentListener`
- `jess.awt.ComponentListener`
- `jess.awt.ContainerListener`
- `jess.awt.FocusListener`
- `jess.awt.ItemListener`
- `jess.awt.KeyListener`
- `jess.awt.MouseListener`
- `jess.awt.MouseMotionListener`
- `jess.awt.TextListener`
- `jess.awt.WindowListener`

Each of these classes implements one of the `Listener` interfaces from the `java.awt.event` package in Java 1.1 and later. Each implementation packages up any event notifications it receives and forwards them to a Jess function, which is supplied as a constructor argument to the `Listener` object.

An example should clarify matters. Let's say that when the `Hello` button is pressed, you would like the string `Hello, World!` to be printed to standard output (how original!). What you need to do is:

1. Define a `deffunction` which prints the message. The `deffunction` will be called with one argument: the event object that would be passed to `actionPerformed()`. (If this is gibberish to you, pick up a book on Java AWT programming.)
2. Create a `jess.awt.ActionListener` object, telling it about this `deffunction`, and also which Jess engine it belongs to. You simply use Jess' `new` command to do this.
3. Tell the `Button` about the `ActionListener` using the `addActionListener` method of `java.awt.Button`.

Here's a complete program in Jess:

```
Jess> ;; Create the widgets
  (defglobal ?*f* = (new java.awt.Frame "Button Demo"))
  Jess> (defglobal ?*b* = (new java.awt.Button "Hello"))

  Jess> ;; Define the deffunction
  (deffunction say-hello "Unconditionally print a message" (?evt)
    (printout t "Hello, World!" crlf))

  Jess> ;; Connect the deffunction to the button
  (?*b* addActionListener
    (new jess.awt.ActionListener say-hello (engine)))

  Jess> ;; Assemble and display the GUI
  (?*f* add ?*b*)
  Jess> (?*f* pack)
  Jess> (set ?*f* visible TRUE)
```

The Jess `engine` function returns the `jess.Rete` object in which it is executed, as an external address. You'll have to quit this program using ^C. To fix this, you can add a `WindowListener` which handles `WINDOW_CLOSING` events to the above program:

```
Jess> ;; If the event is a WINDOW_CLOSING event, exit the program
  (deffunction frame-handler (?evt)
    (if (= (?evt getID) (get-member ?evt WINDOW_CLOSING)) then
        (call (get ?evt source) dispose)
        (exit)))

  Jess>  ;; Connect this deffunction to the frame
  (?*f* addWindowListener
    (new jess.awt.WindowListener frame-handler (engine)))
```

Now when you close the window Jess will exit. Notice how we can examine the `?evt` parameter for event information.

We have used the "raw" AWT widgets here, but this same technique works fine with Swing as well (the new GUI toolkit for Java 1.2).

```
Jess> (defglobal ?*f* = (new javax.swing.JFrame "Button Demo"))
  Jess> (defglobal ?*b* = (new javax.swing.JButton "Hello"))
  Jess> (defglobal ?*p* = (get ?*f* "contentPane"))

  Jess> (deffunction say-hello (?evt)
    (printout t "Hello, World!" crlf))

  Jess> (call ?*b* addActionListener
    (new jess.awt.ActionListener say-hello (engine)))

  Jess> (call ?*p* add ?*b*)
  Jess> (call ?*f* pack)
  Jess> (set ?*f* visible TRUE)

  Jess> (deffunction frame-handler (?evt)
```

```
    (if (= (?evt getID) (get-member ?evt WINDOW_CLOSING)) then
        (call (get ?evt source) dispose)
        (exit)))

  Jess> (?*f* addWindowListener
    (new jess.awt.WindowListener frame-handler (engine)))
```

See the demo `examples/frame.clp` for a slightly more complex example of how you can build an entire Java graphical interface from within Jess.

7.2. Screen Painting and Graphics

As you may know, the most common method of drawing pictures in Java is to subclass `java.awt.Canvas`, overriding the `void paint(Graphics g)` method to call the methods of the `java.awt.Graphics` argument to do the drawing. Well, Jess can't help you to subclass a Java class (at least not yet!), but it does provide an adaptor class, much like the event adaptors described above, that will help you draw pictures. The class is named `jess.awt.Canvas`, and it is a subclass of `java.awt.Canvas.` As such it can be used as a normal Java GUI component. When you construct an instance of this class, you pass in the name of a Jess function and a reference to the Rete engine. Whenever `paint()` is called to render the `jess.awt.Canvas,` the `jess.awt.Canvas` in turn will call the given function. The function will be passed two arguments: the `jess.awt.Canvas` instance itself, and the `java.awt.Graphics` argument to `paint().` In this way, Jess code can draw pictures using Java calls. An example looks like this:

```
Jess> ;; A painting deffunction. This function draws a red 'X' between the
  ;; four corners of the Canvas on a blue field.

  (deffunction painter (?canvas ?graph)
    (bind ?x (get-member (call ?canvas getSize) width))
    (bind ?y (get-member (call ?canvas getSize) height))
    (?graph setColor (get-member java.awt.Color blue))
    (?graph fillRect 0 0 ?x ?y)
    (?graph setColor (get-member java.awt.Color red))
    (?graph drawLine 0 0 ?x ?y)
    (?graph drawLine ?x 0 0 ?y))

  Jess> ;; Create a canvas and install the paint routine.
  (bind ?c (new jess.awt.Canvas painter (engine)))
```

A simple but complete program built on this example is in the file `examples/draw.clp` in the Jess distribution.

Back to index

8. The Jess Function List

In this chapter, every Jess language function shipped with Jess version 6.1 is described. Some of these functions are intrinsic functions while others are optional and may not be available to all Jess code. All of these functions are installed into the command-line version of Jess; to use a function marked (optional) in your own programs, you need to either add the appropriate `Userpackage` or load the script library (see the chapters on Jess/Java programming and extending Jess with Java). The package for each function is listed below.

> *Note:* many functions documented as requiring a specific minimum number of arguments will actually return sensible results with fewer; for example, the + function will return the value of a single argument as its result. This behavior is to be regarded as undocumented and unsupported. In addition, all functions documented as requiring a specific number of arguments will not report an error if invoked with more than that number; extra arguments are simply ignored.

8.1. (- <numeric-expression> <numeric-expression>+)

Package:
    Intrinsics
Arguments:
    Two or more numeric expressions
Returns:
    Number
Description:
    Returns the first argument minus all subsequent arguments. The return value is an `INTEGER` unless any of the arguments are `FLOAT`, in which case it is a `FLOAT`.

8.2. (/ <numeric-expression> <numeric-expression>+)

Package:
    Intrinsics
Arguments:
    Two or more numeric expressions
Returns:
    Number
Description:
    Returns the first argument divided by all subsequent arguments. The return value is a `FLOAT`.

8.3. (* <numeric-expression> <numeric-expression>+)

Package:
    Intrinsics
Arguments:
    Two or more numeric expressions
Returns:
    Number

Description:

Returns the products of its arguments. The return value is an `INTEGER` unless any of the arguments are `FLOAT`, in which case it is a `FLOAT`.

8.4. (** <numeric-expression> <numeric-expression>)

Package:

MathFunctions

Arguments:

Two numeric expressions

Returns:

Number

Description:

Raises its first argument to the power of its second argument (using Java's `Math.pow()` function). ***Note*:** the return value is `NaN` (not a number) if both arguments are negative.

8.5. (+ <numeric-expression> <numeric-expression>+)

Package:

Intrinsics

Arguments:

Two or more numeric expressions

Returns:

Number

Description:

Returns the sum of its arguments. The return value is an `INTEGER` unless any of the arguments are `FLOAT`, in which case it is a `FLOAT`.

8.6. (< <numeric-expression> <numeric-expression>+)

Package:

Intrinsics

Arguments:

Two or more numeric expressions

Returns:

Boolean

Description:

Returns `TRUE` if each argument is less in value than the argument following it; otherwise, returns `FALSE`.

8.7. (<= <numeric-expression> <numeric-expression>+)

Package:

Intrinsics

Arguments:

Two or more numeric expressions

Returns:
Boolean
Description:
Returns TRUE if the value of each argument is less than or equal to the value of the argument following it; otherwise, returns FALSE.

8.8. (<> <numeric-expression> <numeric-expression>+)

Package:
Intrinsics
Arguments:
Two or more numeric expressions
Returns:
Boolean
Description:
Returns TRUE if the value of the first argument is not equal in value to all subsequent arguments; otherwise returns FALSE.

8.9. (= <numeric-expression> <numeric-expression>+)

Package:
Intrinsics
Arguments:
Two or more numeric expressions
Returns:
Boolean
Description:
Returns TRUE if the value of the first argument is equal in value to all subsequent arguments; otherwise, returns FALSE. The integer 2 and the float 2.0 are =, but not eq.

8.10. (> <numeric-expression> <numeric-expression>+)

Package:
Intrinsics
Arguments:
Two or more numeric expressions
Returns:
Boolean
Description:
Returns TRUE if the value of each argument is greater than that of the argument following it; otherwise, returns FALSE.

8.11. (>= <numeric-expression> <numeric-expression>+)

Package:
    Intrinsics
Arguments:
    Two or more numeric expressions
Returns:
    Boolean
Description:
    Returns TRUE if the value of each argument is greater than or equal to that of the argument following it; otherwise, returns FALSE.

8.12. (abs <numeric-expression>)

Package:
    MathFunctions
Arguments:
    One numeric expression
Returns:
    Number
Description:
    Returns the absolute value of its only argument.

8.13. (agenda)

Package:
    MiscFunctions
Arguments:
    Optionally, a module name or the atom *
Returns:
    NIL
Description:
    Displays a list of rule activations to the WSTDOUT router. If no argument is specified, the activations in the current module (not the focus module) are displayed. If a module name is specified, only the activations in that module are displayed. If "*" is specified, then all activations are displayed.

8.14. (and <expression>+)

Package:
    Intrinsics
Arguments:
    One or more expressions
Returns:
    Boolean
Description:
    Returns TRUE if all arguments evaluate to a non-FALSE value; otherwise, returns FALSE.

8.15. (apply <expression>+)

Package:
LispFunctions
Arguments:
One or more expressions
Returns:
An expression
Description:
Returns the result of calling the first argument, as a Jess function, on all the remaining arguments. The strength of this method lies in the fact that you can call a function whose name, for instance, is in a Jess variable.

8.16. (asc <string>)

Package:
Intrinsics
Arguments:
Any string or symbol
Returns:
Integer
Description:
Returns the Unicode value of the first character of the argument, as an RU.INTEGER.

8.17. (assert <fact>+)

Package:
Intrinsics
Arguments:
One or more facts (not fact-IDs)
Returns:
Fact-ID or FALSE
Description:
Adds a fact to the fact list. Asserts all facts onto the fact list; returns the fact-ID of last fact asserted or FALSE if no facts were successfully asserted (for example, if all facts given are duplicates of existing facts.) A JessEvent of type FACT will be sent if the event mask is set appropriately.

8.18. (assert-string <string-expression>)

Package:
Intrinsics
Arguments:
One string representing a fact
Returns:
Fact-ID or FALSE

Description:

Converts a string into a fact and asserts it. Attempts to parse string as a fact, and if successful, returns the value returned by assert with the same fact. Note that the string must contain the fact's enclosing parentheses.

8.19. (bag <bag-command> <bag-arguments>+)

Package:

BagFunctions

Arguments:

An atom (a sub-command) and one or more additional arguments

Returns:

(Varies)

Description:

The `bag` command lets you manipulate Java hashtables from Jess. The net result is that you can create any number of associative arrays or property lists. Each such array or list has a name by which it can be looked up. The lists can contain other lists as properties, or any other Jess data type.

The `bag` command does different things based on its first argument. It's really seven commands in one:

- `create` accepts a String, the name of a new Bag to be created. The bag object itself is returned. For example:

```
Jess> (bag create my-bag)
```

- `delete` accepts the name of an existing bag, and deletes it from the list of bags.
- `find` accepts the name of a bag, and returns the corresponding bag object, if one exists, or `nil.`
- `list` returns a list of the names of all the existing bags, as a multifield.
- `set` accepts as arguments a bag, a String property name, and any Jess value as its three arguments. The named property of the given bag is set to the value, and the value is returned.
- `get` accepts as arguments a bag and a String property name. The named property is retrieved and returned, or `nil` if there is no such property. For example:

```
Jess> (defglobal ?*bag* = 0)
                TRUE
                Jess> (bind ?*bag* (bag create my-bag))
                <External-Address:java.util.Hashtable>
                Jess> (bag set ?*bag* my-prop 3.0)
                3.0
                Jess> (bag get ?*bag* my-prop)
                3.0
```

- `props` accepts a bag as the single argument and returns a multifield consisting of a list of the names of all the properties of that bag.

8.20. (batch <filename>)

Package:
Miscfunctions
Arguments:
One string or atom representing the name of a file
Returns:
(Varies)
Description:
Attempts to parse and evaluate the given file as Jess code. If successful, returns the return value of the last expression in the file.

Note: the argument must follow Jess' rules for valid atoms or strings. On UNIX systems, this presents no particular problems, but Win32 filenames may need special treatment. In particular: pathnames should use either '\\' (double backslash) or '/' (forward slash) instead of '\' (single backslash) as directory separators; and pathnames which include a colon (':') or a space character (' ') *must* be enclosed in double quotes.

In an applet, batch will try to find the file relative to the applet's document base. In any program, if the file is not found, the name is then passed to `ClassLoader.getSystemResourceAsStream()`. This allows files along the class path, including files in JARs, to be batched.

8.21. (bind <variable> <expression>*)

Package:
Intrinsics
Arguments:
A variable name and any value
Returns:
(Varies)
Description:
Binds a variable to a new value. Assigns the given value to the given variable, creating the variable if necessary. Returns the given value.

8.22. (bit-and <integer-expression>+)

Package:
MiscFunctions
Arguments:
One or more integer expressions
Returns:
int
Description:
Performs the bitwise AND of the arguments. `(bit-and 7 4)` is 4, and is equivalent to the Java `7 & 4`.

8.23. (bit-not <integer-expression>)

Package:
MiscFunctions
Arguments:
One integer expression
Returns:
int
Description:
Performs the bitwise NOT of the argument. `(bit-not 0)` is -1, and is equivalent to the Java `~0`.

8.24. (bit-or <integer-expression>+)

Package:
MiscFunctions
Arguments:
One or more integer expressions
Returns:
int
Description:
Performs the bitwise OR of the arguments. `(bit-or 2 4)` is 6, and is equivalent to the Java `2 | 4`.

8.25. (bload <filename>)

Package:
DumpFunctions
Arguments:
One string or atom representing the name of a file
Returns:
TRUE
Description:
The argument is the name of a file previously produced by the `bsave` command. The file is decompressed and deserialized to restore the state of the current Rete object. I/O routers are not restored from the file; they retain their previous state. Furthermore, JessListeners are not restored from the file; again, they are retained from their state prior to the bload.

8.26. (bsave <filename>)

Package:
DumpFunctions
Arguments:
One string or atom representing the name of a file
Returns:
TRUE

Description:

Dumps the engine in which it is called to the given filename argument in a format that can be read using `bload`. Any input/output streams and event listeners are not saved during the serialization process.

8.27. (build <string-expression>)

Package:

Miscfunctions

Arguments:

One string representing some Jess code

Returns:

(Varies)

Description:

Evaluates a string as though it were entered at the command prompt. Only allows constructs to be evaluated. Attempts to parse and evaluate the given string as Jess code. If successful, returns the return value of the last expression in the string. This is typically used to define rules from Jess code. For instance:

```
(build "(defrule foo (foo) => (bar))")
```

Note: The string must consist of one single construct; multiple constructs can be built using multiple calls to build.

8.28. (call (<external-address> | <string-expression>) <string-expression> <call-arguments>+)

Package:

ReflectFunctions

Arguments:

an external address or String, a String, and any number of additional arguments

Returns:

(Varies)

Description:

Calls a Java method on the given object, or a static method of the class named by the first argument. The second argument is the name of the method, and subsequent arguments are passed to the method. Arguments are promoted and overloaded methods selected precisely as for `new`. The return value is converted to a suitable Jess value before being returned. Array return values are converted to multifields.

The functor `call` may be omitted if the method being called is non-static. The following two method calls are equivalent:

```
;; These are legal and equivalent
(call ?vector addElement (new java.lang.String "Foo"))
(?vector addElement (new java.lang.String "Foo"))
```

Note that `call` may now be omitted if the object comes from the return value of another function

call:

```
;; This is now legal
((new java.lang.Vector 10) addElement (new java.lang.String "Foo"))
```

8.29. (call-on-engine <external-address> <jess-code>)

Package:
MiscFunctions
Arguments:
an external address (must be a jess.Rete object), and an executable snippet of Jess code
Returns:
(Varies)
Description:
Executes some Jess code in the context of the given Rete object. This is a nice way to send messages between multiple Rete engines in one process. Note that the current variable context is used to evaluate the code, so (for instance) all defglobal values will be from the calling engine, not the target.

8.30. (clear)

Package:
Intrinsics
Arguments:
None
Returns:
TRUE
Description:
Clears Jess. Deletes all rules, deffacts, defglobals, templates, facts, activations, and so forth. Java Userfunctions are not deleted.

8.31. (clear-focus-stack)

Package:
ModuleFunctions
Arguments:
None
Returns:
NIL
Description:
Removes all modules from the focus stack.

8.32. (clear-storage)

Package:
Intrinsics
Arguments:
None

Returns:
TRUE
Description:
Clears the hashtable used by `(store)` and `(fetch)`.

8.33. (close [<router-identifier>])

Package:
Intrinsics
Arguments:
One or more router identifiers (atoms)
Returns:
TRUE
Description:
Closes any I/O routers associated with the given name by calling `close()` on the underlying stream, then removes the routers. Any subsequent attempt to use a closed router will report `bad router`. See open.

8.34. (complement$ <multifield-expression> <multifield-expression>)

Package:
MultiFunctions
Arguments:
Two multifields
Returns:
Multifield
Description:
Returns a new multifield consisting of all elements of the second multifield not appearing in the first multifield.

8.35. (context)

Package:
ReflectFunctions
Arguments:
None
Returns:
External address
Description:
Returns the execution context (a jess.Context object) it is called in. This provides a way for deffunctions to get a handle to this useful class.

8.36. (count-query-results <query-name> <expression>+)

Package:
MiscFunctions
Arguments:
An atom, and zero or more additional expressions
Returns:
INTEGER
Description:
Runs a query and returns a count of the matches. See the documentation for defquery for more details. Also see run-query for caveats concerning calling count-query-results on a rule RHS.

8.37. (create$ <expression>*)

Package:
MultiFunctions
Arguments:
Zero or more expressions
Returns:
Multifield
Description:
Appends its arguments together to create a multifield value. Returns a new multifield containing all the given arguments, in order. For each argument that is a multifield, the individual elements of the multifield are added to the new multifield; this function will not create nested multifields (which are not meaningful in the Jess language.) ***Note***: multifields must be created explicitly using this function or others that return them. Multifields cannot be directly parsed from Jess input.

8.38. (defadvice (before | after) (<function-name> | <multifield>) <function-call>+)

Package:
Intrinsics
Arguments:
The atom `before` or the atom `after`, followed by either one function name or a multifield of function names or the atom `ALL`, followed by one or more function calls.
Returns:
(varies)
Description:
Lets you supply extra code to run before or after the named function(s) or all functions. If `before` is specified, the code will execute before the named function(s); the variable $?argv will hold the entire function call vector (function name and parameters) on entry to and exit from the code block. If `after` is specified, the function will be called before the code block is entered. When the block is entered, the variable ?retval will refer to the original function's return value.

Whether `before` or `after` is specified, if the code block explicitly calls `return` with a value, the returned value will appear to the caller to be the return value of the original function. For `before` advice, this means the original function will not be called in this case.

8.39. (defclass <tag> <Java class name> [extends <tag>])

Package:
ReflectFunctions
Arguments:
Two or four atoms, as noted above
Returns:
The second argument
Description:
Defines a template with the given tag, with slots based on the Java Beans properties found in the named class. If the optional extends clause is included, the second tag will become the parent template of the new template. The common slots in the two templates will be in the same order, at the beginning of the new template. Rules defined to match instances of the parent template will also match instances of the new child template.

8.40. (definstance <tag> <Java object> [static | dynamic])

Package:
ReflectFunctions
Arguments:
An atom, a Java object, and (optionally) one of the atoms static or dynamic.
Returns:
The fact-id of the new shadow fact.
Description:
Creates a "shadow fact" representing the given Java object, according to the named template (which should have come from `defclass`.) If the atom `static` is not supplied as the optional third argument, a `PropertyChangeListener` is installed in the given object, so that Jess can keep the shadow fact updated if the object's properties change.

Note that it is an error for a given Java object to be installed in more than one `definstance` at a time. The second and subsequent `definstance` calls for a given object will return a fact-id with value -1.

8.41. (delete$ <multifield-expression> <begin-integer-expression> <end-integer-expression>)

Package:
MultiFunctions
Arguments:
A multifield and two integer expressions
Returns:
Multifield
Description:
Deletes the specified range from a multifield value. The first numeric expression is the 1-based index of the first element to remove; the second is the 1-based index of the last element to remove.

8.42. (dependencies <fact-id>)

Package:
    MiscFunctions
Arguments:
    A Fact or fact-id
Returns:
    A list
Description:
    Returns a list containing all the `jess.Token` objects that give logical support from the argument fact; if there are none this function returns an empty list. A `jess.Token` object is a list of facts; they're the same objects returned by `run-query`.

8.43. (dependents <fact-id>)

Package:
    MiscFunctions
Arguments:
    A Fact or fact-id
Returns:
    A list
Description:
    Returns a list containing all the `jess.Fact` objects that get logical support from the argument fact; if there are none this function returns an empty list.

8.44. (div <numeric-expression> <numeric-expression>+)

Package:
    MathFunctions
Arguments:
    Two or more numeric expressions
Returns:
    Numbers
Description:
    Returns the first argument divided by all subsequent arguments using integer division. Quotient of the values of the two numeric expressions rounded to the nearest integer.

8.45. (do-backward-chaining <deftemplate-tag>)

Package:
    Intrinsics
Arguments:
    Name of a template (ordered or unordered)
Returns:
    TRUE

Description:
Marks a template as being eligible for backwards chaining, as described in the text. If the template is unordered -- i.e., if it is explicitly defined with a (deftemplate) construct -- then it must be defined *before* calling `do-backward-chaining`. In addition, this function must be called *before* defining any rules which use the template.

8.46. (duplicate <fact-specifier> (<slot-name> <value>)+)

Package:
Intrinsics
Arguments:
A fact-ID and one or more two-element lists
Returns:
Fact-ID
Description:
Makes a copy of the fact; the fact-ID must belong to an unordered fact. Each list is taken as the name of a slot in this fact and a new value to assign to the slot. A new fact is asserted which is similar to the given fact but which has the specified slots replaced with new values. The fact-ID of the new fact is returned. It is an error to call `duplicate` on a definstance.

8.47. (e)

Package:
MathFunctions
Arguments:
None
Returns:
Number
Description:
Returns the transcendental number *e*.

8.48. (engine)

Package:
MiscFunctions
Arguments:
None
Returns:
External address
Description:
Returns an external-address object containing the Rete engine in which the function in called.

8.49. (eq <expression> <expression>+)

Package:
    Intrinsics
Arguments:
    Two or more arbitrary arguments
Returns:
    Boolean
Description:
    Returns `TRUE` if the first argument is equal in type and value to all subsequent arguments. For strings, this means identical contents. Uses the Java `Object.equals()` function, so can be redefined for external types. Note that the integer `2` and the floating-point number `2.0` are *not* `eq`, but they are `eq*` and `=`.

8.50. (eq* <expression> <expression>+)

Package:
    Intrinsics
Arguments:
    Two or more arbitrary arguments
Returns:
    Boolean
Description:
    Returns `TRUE` if the first argument is equivalent to all the others. Uses numeric equality for numeric types, unlike `eq`. Note that the integer 2 and the floating-point number 2.0 are *not* `eq`, but they are `eq*` and `=`.

8.51. (eval <lexeme-expression>)

Package:
    Intrinsics
Arguments:
    One string containing a valid Jess expression
Returns:
    (Varies)
Description:
    Evaluates a string as though it were entered at a command prompt. Only allows functions to be evaluated. Evaluates the string as if entered at the command line and returns the result.

    Note: The string must consist of one single function call; multiple calls can be evaluated using multiple calls to eval.

8.52. (evenp <expression>)

Package:
PredFunctions
Arguments:
One numeric expression
Returns:
Boolean
Description:
Returns TRUE for even numbers; otherwise, returns FALSE. Results with non-integers may be unpredictable.

8.53. (exit)

Package:
Intrinsics
Arguments:
None
Returns:
Nothing
Description:
Exits Jess and halts Java.

8.54. (exp <numeric-expression>)

Package:
MathFunctions
Arguments:
One numeric expression
Returns:
Number
Description:
Raises the value *e* to the power of its only argument.

8.55. (explode$ <string-expression>)

Package:
MultiFunctions
Arguments:
One string
Returns:
Multifield
Description:
Creates a multifield value from a string. Parses the string as if by a succession of `read` calls, then returns these individual values as the elements of a multifield.

8.56. (external-addressp <expression>)

Package:
PredFunctions
Arguments:
One expression
Returns:
Boolean
Description:
Returns TRUE or FALSE as the given expression is an external-address.

8.57. (fact-id <integer>)

Package:
Miscfunctions
Arguments:
One number, a fact-id
Returns:
The given number as an RU.FACT
Description:
If the argument is the fact-id of an existing fact, returns the number as a value of type RU.FACT; otherwise throws an exception.

8.58. (facts)

Package:
Scriptlib
Arguments:
None
Returns:
TRUE
Description:
Prints a list of all facts on the fact list.

8.59. (fact-slot-value <fact-id> <slot-name>)

Package:
Scriptlib
Arguments:
A fact-id and a slot name
Returns:
(varies)
Description:
Returns the value in the named slot of the fact with the given fact-id.

8.60. (fetch <string or atom>)

Package:
    Intrinsics
Arguments:
    One string or atom
Returns:
    (varies)
Description:
    Retrieves and returns any value previously stored by the `store` function under the given name, or `nil` if there is none. Analogous to the `fetch()` member function of the `Rete` class. See the section on using store and fetch for details.

8.61. (first$ <multifield-expression>)

Package:
    MultiFunctions
Arguments:
    One multifield
Returns:
    Multifield
Description:
    Returns the first field of a multifield as a new 1-element multifield.

8.62. (float <numeric-expression>)

Package:
    MathFunctions
Arguments:
    One numeric expression
Returns:
    Floating-point number
Description:
    Converts its only argument to a float.

8.63. (floatp <expression>)

Package:
    PredFunctions
Arguments:
    One numeric expression
Returns:
    Boolean
Description:
    Returns TRUE for floats; otherwise, returns FALSE.

8.64. (focus <module-name>+)

Package:
ModuleFunctions
Arguments:
One or more atoms, the names of modules
Returns:
The name of the previous focus module
Description:
Changes the focus module. The next time the engine runs, the first rule to fire will be from the first module listed (if any rules are activated in this module.) The previously active module is pushed down on the focus stack. If more than one module is listed, they are pushed onto the focus stack in order from right to left.

8.65. (foreach <variable> <multifield-expression> <action>*)

Package:
Intrinsics
Arguments:
A variable, a multifield expression, and zero or more arguments
Returns:
Varies
Description:
The named variable is set to each of the values in the multifield in turn; for each value, all of the other arguments are evaluated in order. The `return` function can be used to break the iteration.

Example:

```
(foreach ?x (create$ a b c d) (printout t ?x crlf))
```

8.66. (format <router-identifier> <string-expression> <expression>*)

Package:
MiscFunctions
Arguments:
A router identifier, a format string, and zero or more arguments
Returns:
A string
Description:
Sends formatted output to the specified logical name. Formats the arguments into a string according to the format string, which is identical to that used by `printf` in the C language (find a C book for more information). Returns the string, and optionally prints the string to the named router. If you pass `nil` for the router name, no printing is done.

8.67. (gensym*)

Package:
    Intrinsics
Arguments:
    None
Returns:
    Atom
Description:
    Returns a special unique sequenced value. Returns a unique atom which consists of the letters `gen` plus an integer. Use `setgen` to set the value of the integer to be used by the next gensym call.

8.68. (get <external-address> <string-expression>)

Package:
    ReflectFunctions
Arguments:
    An external address and a string.
Returns:
    (Varies)
Description:
    Retrieves the value of a Java Bean's property. The first argument is the object and the second argument is the name of the property. The return value is converted to a suitable Jess value exactly as for call.

8.69. (get-current-module)

Package:
    ModuleFunctions
Arguments:
    None
Returns:
    An atom, the name of the current module
Description:
    Gets the current module (see set-current-module).

8.70. (get-focus)

Package:
    ModuleFunctions
Arguments:
    None
Returns:
    Atom
Description:
    Returns the name of the current focus module (see focus).

8.71. (get-focus-stack)

Package:
ModuleFunctions
Arguments:
None
Returns:
Multifield
Description:
Returns the module names on the focus stack as a multifield. The top module on the stack is the first entry in the multifield.

8.72. (get-member (<external-address> | <string-expression>) <string-expression>)

Package:
reflect,ReflectFunctions
Arguments:
An external address or a string, and a string.
Returns:
(Varies)
Description:
Retrieves the value of a Java object's data member. The first argument is the object (or the name of a class, for a static member) and the second argument is the name of the field. The return value is converted to a suitable Jess value exactly as for call.

8.73. (get-multithreaded-io)

Package:
Intrinsics
Arguments:
None
Returns:
Boolean
Description:
Returns TRUE is Jess is currently using a separate thread to flush I/O streams. Turning this on can lead to a modest performance enhancement, at the expense of possible loss of output on program termination.

8.74. (get-reset-globals)

Package:
MiscFunctions
Arguments:
None
Returns:
Boolean

Description:
Indicates the current setting of global variable reset behavior. See set-reset-globals for an explanation of this property.

8.75. get-salience-evaluation

Package:
MiscFunctions
Arguments:
None
Returns:
Atom
Description:
Indicates the current setting of salience evaluation behavior. See set-salience-evaluation for an explanation of this property.

8.76. (get-strategy)

Package:
MiscFunctions
Arguments:
(None)
Returns:
An atom, the name of the current conflict resolution strategy.
Description:
Returns the name of the current conflict resolution strategy. See `set-strategy`.

8.77. (halt)

Package:
Intrinsics
Arguments:
None
Returns:
TRUE
Description:
Halts rule execution. No effect unless called from the RHS of a rule.

8.78. (if <expression> then <action>* [else <action>*])

Package:
Intrinsics
Arguments:
A Boolean variable or function call returning Boolean, the atom then, and any number of additional expressions; optionally followed by the atom else another list of expression.

Returns:
(Varies)
Description:
Allows conditional execution of a group of actions. The boolean expression is evaluated. If it does not evaluate to FALSE, the first list of expressions is evaluated, and the return value is that returned by the last expression of that list. If it does evaluate to FALSE, and the optional second list of expressions is supplied, those expressions are evaluated and the value of the last is returned.

Example:

```
(if (> ?x 100)
then
(printout t "X is big" crlf)
else
(printout t "X is small" crlf))
```

8.79. (implode$ <multifield-expression>)

Package:
MultiFunctions
Arguments:
One multifield
Returns:
String
Description:
Creates a string from a multifield value. Converts each element of the multifield to a string, and returns these strings concatenated with single intervening spaces.

8.80. (import <atom>)

Package:
ReflectFunctions
Arguments:
One atom
Returns:
TRUE
Description:
Works like the Java `import` statement. You can import either a whole package using

```
(import java.io.*)
```

or a single class using

```
(import java.awt.Button)
```

After that, all functions that can accept a Java class name (`new`, `defclass`, `call`, etc) will refer to the import list to try to find the class that goes with a specific name. Note that `java.lang.*` is now implicitly imported.

8.81. (insert$ <multifield-expression> <integer-expression> <single-or-multifield-expression>+)

Package:
MultiFunctions
Arguments:
A multifield, an integer, and one or more multifields
Returns:
A multifield
Description:
Inserts one or more values in a multifield. Inserts the elements of the second and later multifields so that they appear starting at the given 1-based index of the first multifield.

8.82. (instanceof <external-address> <class-name>)

Package:
ReflectFunctions
Arguments:
An external address and the name of a Java class
Returns:
Boolean
Description:
Returns true if the EXTERNAL_ADDRESS object can be assigned to a variable whose type is given by the class name. Implemented using java.lang.Class.isInstance() . The class name can be fully-qualified or it can be an imported name; see the discussion of the `import` function.

8.83. (integer <numeric-expression>)

Package:
MathFunctions
Arguments:
One numeric expression
Returns:
Integer
Description:
Converts its only argument to an integer. Truncates any fractional component of the value of the given numeric expression and returns the integral part.

8.84. (integerp <expression>)

Package:
PredFunctions
Arguments:
One expression
Returns:
Boolean

Description:
Returns TRUE for integers; otherwise, returns FALSE.

8.85. (intersection$ <multifield-expression> <multifield-expression>)

Package:
MultiFunctions
Arguments:
Two multifields
Returns:
Multifield
Description:
Returns the intersection of two multifields. Returns a multifield consisting of the elements the two argument multifields have in common.

8.86. (jess-version-number)

Package:
Intrinsics
Arguments:
None
Returns:
Float
Description:
Returns a version number for Jess; currently 6.1 .

8.87. (jess-version-string)

Package:
Intrinsics
Arguments:
None
Returns:
String
Description:
Returns a human-readable string descriptive of this version of Jess.

8.88. (length$ <multifield-expression>)

Package:
MultiFunctions
Arguments:
Multifield
Returns:
Integer

Description:
Returns the number of fields in a multifield value.

8.89. (lexemep <expression>)

Package:
PredFunctions
Arguments:
Any expression
Returns:
Boolean
Description:
Returns TRUE for symbols and strings; otherwise, returns FALSE.

8.90. (list-deftemplates [*|module-name])

Package:
MiscFunctions
Arguments:
Optionally, a module name, or the atom *
Returns:
nil
Description:
With no arguments, prints a list of all deftemplates in the current module (not the focus module) to the ’t’ router. With a module name for an argument, prints the names of the templates in that module. With "*" as an argument, prints the names of all templates.

8.91. (list-focus-stack)

Package:
ModuleFunctions
Arguments:
None
Returns:
NIL
Description:
Displays the module focus stack, one module per line; the top of the stack (the focus module) is displayed first.

8.92. (list-function$)

Package:
Intrinsics
Arguments:
None

Returns:
Multifield
Description:
Returns a multifield list of all the functions currently callable, including intrinsics, deffunctions, and Userfunctions. Each function name is an atom. The names are sorted in alphabetical order.

8.93. (load-facts <file-name>)

Package:
Intrinsics
Arguments:
A string or atom representing the name of a file of facts
Returns:
Boolean
Description:
Asserts facts loaded from a file. The argument should name a file containing a list of facts (not deffacts constructs, and no other commands or constructs). Jess will parse the file and assert each fact. The return value is the return value of assert when asserting the last fact. In an applet, `load-facts` will use `getDocumentBase()` to find the named file.

Note: See the `batch` command for a discussion about specifying filenames in Jess.

8.94. (load-function <class-name>)

Package:
MiscFunctions
Arguments:
One string or atom representing the name of a Java class
Returns:
Boolean
Description:
The argument must be the fully-qualified name of a Java class that implements the Userfunction interface. The class is loaded in to Jess and added to the engine, thus making the corresponding command available. See Extending Jess with Java for more information.

8.95. (load-package <class-name>)

Package:
MiscFunctions
Arguments:
One string or atom, the name of a Java class
Returns:
Boolean
Description:
The argument must be the fully-qualified name of a Java class that implements the Userpackage interface. The class is loaded in to Jess and added to the engine, thus making the corresponding package of commands available. See Extending Jess with Java for more information.

8.96. (log <numeric-expression>)

Package:
MathFunctions
Arguments:
One numeric expression
Returns:
Number
Description:
Returns the logarithm base e of its only argument.

8.97. (log10 <numeric-expression>)

Package:
MathFunctions
Arguments:
One numeric expression
Returns:
Number
Description:
Returns the logarithm base-10 of its only argument.

8.98. (long <expression>)

Package:
MiscFunctions
Arguments:
One expression, either numeric or String
Returns:
RU.LONG
Description:
Interprets the expression as a Java long (if possible) and returns a long value. Use strings or atoms for precise values that can’t be expressed as an int. Longs in Jess are "second class citizens" in the sense that you can’t directly do math on them. You can assign them to variables, pass them to function calls, and convert them to Strings or floating-point numbers.

8.99. (longp <expression>)

Package:
MiscFunctions
Arguments:
One expression
Returns:
Boolean
Description:
Returns TRUE if the expression is of type RU.LONG; FALSE otherwise.

8.100. (lowcase <lexeme-expression>)

Package:
StringFunctions
Arguments:
One atom or string.
Returns:
String
Description:
Converts uppercase characters in a string or symbol to lowercase. Returns the argument as an all-lowercase string.

8.101. (matches <lexeme-expression>)

Package:
ViewFunctions
Arguments:
One atom, a rule or query name
Returns:
TRUE
Description:
Produces an ugly printout, useful for debugging, of the contents of the left and right Rete memories of each two-input node on the given rule or query's LHS.

8.102. (max <numeric-expression>+)

Package:
MathFunctions
Arguments:
One or more numerical expressions
Returns:
Number
Description:
Returns the value of its largest numeric argument

8.103. (member$ <single-field-expression> <multifield-expression>)

Package:
MultiFunctions
Arguments:
A value and a multifield
Returns:
Integer or FALSE
Description:
Returns the position (1-based index) of a single-field value within a multifield value; otherwise, returns FALSE.

8.104. (min <numeric-expression>+)

Package:
MathFunctions
Arguments:
One or more numeric expressions
Returns:
Number
Description:
Returns the value of its smallest numeric argument.

8.105. (mod <numeric-expression> <numeric-expression>)

Package:
Intrinsics
Arguments:
Two integer expressions
Returns:
Integer
Description:
Returns the remainder of the result of dividing the first argument by its second (assuming that the result of the division must be an integer).

8.106. (modify <fact-specifier> (<slot-name> <value>)+)

Package:
Intrinsics
Arguments:
A fact and one or more two-element lists
Returns:
Fact-ID
Description:
Modifies the given unordered fact in the fact list. The fact must be an unordered fact. Each list is taken as the name of a slot in this fact and a new value to assign to the slot. The fact is removed from the fact list, the values in the specified slots are replaced with the new values, and the fact is reasserted. The fact-ID of the fact does not change. The fact itself is returned. A JessEvent of type FACT + MODIFIED will be sent if the event mask is set appropriately.

Modifying a `definstance` fact will cause the appropriate object properties to be set as well.

8.107. (multifieldp <expression>)

Package:
PredFunctions
Arguments:
Any value

Returns:
Boolean
Description:
Returns TRUE for multifield values; otherwise, returns FALSE.

8.108. (neq <expression> <expression>+)

Package:
Intrinsics
Arguments:
Two or more values
Returns:
Boolean
Description:
Returns TRUE if the first argument is not equal in type and value to all subsequent arguments (see eq).

8.109. (new <string-expression> <new-arguments>+)

Package:
ReflectFunctions
Arguments:
A string and one or more arguments
Returns:
Boolean
Description:
Creates a new Java object and returns an EXTERNAL_ADDRESS value containing it. The first argument is the fully-qualified class name: `java.util.Vector`, for example. The second and later arguments are constructor arguments. The constructor will be chosen from among all constuctors for the named class based on a *first-best fit* algorithm. Built-in Jess types are converted as necessary to match available constructors. See the text for more details.

8.110. (not <expression>)

Package:
Intrinsics
Arguments:
One expression
Returns:
Boolean
Description:
Returns TRUE if its only arguments evaluates to FALSE; otherwise, returns FALSE.

8.111. (nth$ <integer-expression> <multifield-expression>)

Package:
MultiFunctions
Arguments:
A number and a multifield
Returns:
(Varies)
Description:
Returns the value of the specified (1-based index) field of a multifield value.

8.112. (numberp <expression>)

Package:
PredFunctions
Arguments:
One expression
Returns:
Boolean
Description:
Returns TRUE for numbers; otherwise, returns FALSE.

8.113. (oddp <integer-expression>)

Package:
PredFunctions
Arguments:
One integer expression
Returns:
Boolean
Description:
Returns TRUE for odd numbers; otherwise, returns FALSE; see evenp.

8.114. (open <file-name> <router-identifier> [r|w|a])

Package:
Intrinsics
Arguments:
A file name, an identifier for the file (an atom), and optionally a mode string, one of r, w, a.
Returns:
The file identifier, a router name.
Description:
Opens a file. Subsequently, the given router identifier can be passed to printout, read, readline, or any other functions that accept I/O routers as arguments. By default, the file is opened for reading; if a mode string is given, it may be opened for reading only (r), writing only (w), or appending (a).

Note: See the `batch` command for a discussion about specifying filenames in Jess.

8.115. (or <expression>+)

Package:
Intrinsics
Arguments:
One or more expressions
Returns:
Boolean
Description:
Returns `TRUE` if any of the arguments evaluates to a non-`FALSE` value; otherwise, returns `FALSE`.

8.116. (pi)

Package:
MathFunctions
Arguments:
None
Returns:
Number
Description:
Returns the number `pi`.

8.117. (pop-focus)

Package:
ModuleFunctions
Arguments:
None
Returns:
Atom, the name of a module
Description:
Removes the top module from the focus stack and returns its name.

8.118. (ppdeffacts <atom>)

Package:
Scriptlib
Arguments:
An atom, the name of a deffacts
Returns:
String
Description:
Returns a pretty-print rendering of a deffacts.

8.119. (ppdeffunction <atom>)

Package:
    Scriptlib
Arguments:
    An atom, the name of a deffunction
Returns:
    String
Description:
    Returns a pretty-print representation of a deffunction.

8.120. (ppdefglobal <atom>)

Package:
    Scriptlib
Arguments:
    An atom, the name of a defglobal
Returns:
    String
Description:
    Returns a pretty-print representation of a defglobal.

8.121. (ppdefquery <atom>)

Package:
    Scriptlib
Arguments:
    An atom, the name of a defquery
Returns:
    String
Description:
    Returns a pretty-print rendering of a defquery.

8.122. (ppdefrule <atom>)

Package:
    Scriptlib
Arguments:
    An atom, the name of a rule or query
Returns:
    String
Description:
    Returns a pretty-print rendering of a rule or query.

8.123. (ppdeftemplate <atom>)

Package:
    Scriptlib
Arguments:
    An atom, the name of a template
Returns:
    String
Description:
    Returns a pretty-print representation of a template.

8.124. (printout <router-identifier> <expression>*)

Package:
    Intrinsics
Arguments:
    A router identifier followed by zero or more expressions
Returns:
    nil
Description:
    Sends unformatted output to the specified logical name. Prints its arguments to the named router, which must be open for output. No spaces are added between arguments. The special atom `crlf` prints as a newline. The special router name `t` can be used to signify standard output.

8.125. (progn <expression>+)

Package:
    LispFunctions
Arguments:
    One or more expressions
Returns:
    The result of evaluating the last expression.
Description:
    A simple control structure that allows you to group multiple function calls where syntactically only one is allowed - for instance, on the LHS of a rule.

8.126. (random)

Package:
    MathFunctions
Arguments:
    None
Returns:
    Number
Description:
    Returns a pseudo-random integer between 0 and 65536.

8.127. (read [<router-identifier>])

Package:
Intrinsics
Arguments:
An optional input router identifier (when omitted t is the default)
Returns:
(Varies)
Description:
Reads a single-field value from a specified logical name. Read a single atom, string, or number from the named router, returns this value. The router `t` means standard input. Newlines are treated as ordinary whitespace; this behaviour is different than in CLIPS, which returns newlines as tokens. If you need to parse text line-by-line, use `readline` and `explode$`.

8.128. (readline [<router-identifier>])

Package:
Intrinsics
Arguments:
An optional input router identifier (when omitted t is the default)
Returns:
String
Description:
Reads an entire line as a string from the specified logical name (router). The router `t` means standard input.

8.129. (replace$ <multifield-expression> <begin-integer-expression> <end-integer-expression> <multifield-expression>+)

Package:
MultiFunctions
Arguments:
A multifield, two numeric expressions, and one or more additional single or multifield values
Returns:
Multifield
Description:
Replaces the specified range of a multifield value with a set of values. The variable number of final arguments are inserted into the first multifield, replacing elements between the 1-based indices given by the two numeric arguments, inclusive. Example:

```
Jess> (replace$ (create$ a b c) 2 2 (create$ x y z))
           (a x y z c)
```

8.130. (reset)

Package:
    Intrinsics
Arguments:
    None
Returns:
    TRUE
Description:
    Removes all facts from the fact list, removes all activations, then asserts the fact `(initial-fact)`, then asserts all facts found in deffacts, asserts a fact representing each registered definstance, and (if the set-reset-globals property is TRUE) initializes all defglobals.

8.131. (rest$ <multifield-expression>)

Package:
    MultiFunctions
Arguments:
    One multifield
Returns:
    Multifield
Description:
    Returns all but the first field of a multifield as a new multifield.

8.132. (retract <integer-expression>+)

Package:
    Intrinsics
Arguments:
    One or more fact-IDs (*not* integers)
Returns:
    TRUE
Description:
    Retracts the facts whose IDs are given. Retracting a definstance fact will result in an implict call to undefinstance for the corresponding object (the object will no longer be pattern-matched). A JessEvent of type FACT + REMOVED will be sent if the event mask is set appropriately.

8.133. (retract-string <string>)

Package:
    Intrinsics
Arguments:
    A string, a representation of a Fact
Returns:
    TRUE

Description:
Parses the string as a Fact; if such a fact exists on the knowledge base, calls `retract` on it.

8.134. (return [<expression>])

Package:
Intrinsics
Arguments:
An optional expression
Returns:
(Varies)
Description:
From a deffunction, returns the given value and exits the deffunction immediately. From the RHS of a rule, terminates the rule's execution immediately and pops the current focus module from the focus stack. No argument should be given when return is called from the RHS of a rule.

8.135. (round <numeric-expression>)

Package:
MathFunctions
Arguments:
One numeric expression
Returns:
Integer
Description:
Rounds its argument toward the closest integer or negative infinity if exactly between two integers.

8.136. (rules)

Package:
MiscFunctions
Arguments:
Optionally, a module name, or the atom *
Returns:
nil
Description:
With no arguments, prints a list of all rules and queries in the current module (not the focus module) to the 't' router. With a module name for an argument, prints the names of the rules and queries in that module. With "*" as an argument, prints the names of all rules and queries.

8.137. (run [<integer>])

Package:
Intrinsics
Arguments:
Optionally, a single integer

Returns:
TRUE
Description:
Starts the inference engine. If no argument is supplied, Jess will keep running until no more activations remain or `halt` is called. If an argument is supplied, it gives the maximum number of rules to fire before stopping.

8.138. (run-query <query-name> <expression>+)

Package:
MiscFunctions
Arguments:
An atom, and zero or more additional expressions
Returns:
java.util.Iterator, as an EXTERNAL_ADDRESS
Description:
Runs a query and returns a `java.util.Iterator` of the matches. See the documentation for defquery for more details. Note that run-query can lead to backwards chaining, which can cause rules to fire; thus if run-query is called on a rule RHS, other rules' RHSs may run to completion before the instigating rule completes. Putting run-query on a rule RHS can also cause the count of executed rules returned by `run` to be low. Note that the `Iterator` returned by this function should be used immediately. It will become invalid if any of the following functions are called before you've used it: `reset`, `count-query-results`, or `run-query`. It *may* become invalid if any of the following are called: `assert`, `retract`, `modify`, or `duplicate`, and if any of the affected facts are involved in the active query's result.

8.139. (run-until-halt)

Package:
Scriptlib
Arguments:
None.
Returns:
int
Description:
Runs the engine until `halt` is called. Returns the number of rules fired. When there are no active rules, the calling thread will be blocked waiting on the activation semaphore.

8.140. (save-facts <file-name> [<deftemplate-name>])

Package:
Intrinsics
Arguments:
A filename, and optionally an atom
Returns:
Boolean

Description:

Saves facts to a file. Attempts to open the named file for writing, and then writes a list of all facts on the fact list to the file. This file is suitable for reading with load-facts. If the optional second argument is given, only facts whose head matches this atom will be saved. Does not work in applets.

Note: See the `batch` command for a discussion about specifying filenames in Jess.

8.141. (set <external-address> <string-expression> <expression>)

Package:

ReflectFunctions

Arguments:

An external address, a string, and an expression

Returns:

The last argument

Description:

Sets a Java Bean's property to the given value. The first argument is the Bean object; the second argument is the name of the property. The third value is the new value for the property; the same conversions are applied as for `new` and `call`.

8.142. (set-current-module <module-name>)

Package:

ModuleFunctions

Arguments:

An atom, the name of a valid module

Returns:

An atom, the name of the previous current module

Description:

Sets the current module. Any constructs defined without explicitly naming a module are defined in the current module. Note that defining a defmodule also sets the current module.

8.143. (set-factory [factory object])

Package:

MiscFunctions

Arguments:

External address, an object that implements the interface jess.factory.Factory

Returns:

External address, an object that implements the interface jess.factory.Factory; the previous value of this property.

Description:

Set the "thing factory" for the active Rete object. Providing an alternate "thing factory" is a very advanced, and currently undocumented, way to extend Jess's functionality.

8.144. (setgen <numeric-expression>)

Package:
MiscFunctions
Arguments:
A numeric expression
Returns:
TRUE
Description:
Sets the starting number used by `gensym*`. Note that if this number has already been used, `gensym*` uses the next larger number that has not been used.

8.145. (set-member (<external-address> | <string-expression>) <string> <expression>+)

Package:
ReflectFunctions
Arguments:
An external address or a string, a string, and one or more expressions
Returns:
The last argument
Description:
Sets a Java object's member variable to the given value. The first argument is the object (or the name of the class, in the case of a static member variable). The second argument is the name of the variable. The third value is the new value for the variable; the same conversions are applied as for `new` and `call`.

8.146. (set-multithreaded-io (TRUE | FALSE))

Package:
Intrinsics
Arguments:
Boolean
Returns:
Boolean
Description:
Specify whether Jess should use a separate thread to flush I/O streams. Turning this on can lead to a modest performance enhancement, at the expense of possible loss of output on program termination. Returns the previous value of this property.

8.147. (set-node-index-hash <integer>)

Package:
MiscFunctions
Arguments:
One integral value

Returns:
TRUE
Description:
Sets the default hashing key used in all Rete network join node memories defined after the function is called; this function will not affect parts of the network already in existence at the time of the call. A small value will give rise to memory-efficient nodes; a larger value will use more memory. If the created nodes will generally have to remember many partial matches, large numbers will lead to faster performance; the opposite may be true for nodes which will rarely hold more than one or two partial matches. This function sets the default; explicit `declare` statements can override this for specific rules.

8.148. (set-reset-globals (TRUE | FALSE | nil))

Package:
MiscFunctions
Arguments:
One boolean value (TRUE or FALSE or nil)
Returns:
Boolean
Description:
Changes the current setting of the global variable reset behavior. If this property is set to TRUE (the default), then the (reset) command reinitializes the values of global variables to their initial values (if the initial value was a function call, the function call is reexecuted.) If the property is set to FALSE or nil, then (reset) will not affect global variables. Note that in previous versions of Jess, defglobals were always reset; but if the initial value was set with a function call, the function was **not** reevaluated. Now it is.

8.149. (set-salience-evaluation (when-defined | when-activated | every-cycle))

Package:
MiscFunctions
Arguments:
One of the atoms when-defined, when-activated, or every-cycle
Returns:
One of the potential arguments (the previous value of this property)
Description:
Changes the current setting of the salience evaluation behavior. By default, a rule's salience will be determined once, when the rule is defined (when-defined.) If this property is set to when-activated, then the salience of each rule will be redetermined immediately before each time it is placed on the agenda. If the property is set to every-cycle, then the salience of every rule is redetermined immediately after each time any rule fires.

8.150. (set-strategy (depth | breadth))

Package:
MiscFunctions
Arguments:
An atom or string representing the name of a strategy (can be a fully-qualifed Java class name). You can use depth and breadth to represent the two built-in strategies.
Returns:
The previous strategy as an atom.
Description:
Lets you specify the *conflict resolution strategy* Jess uses to order the firing of rules of equal salience. Currently, there are two strategies available: *depth (LIFO)* and *breadth (FIFO).* When the depth strategy is in effect (the default), more recently activated rules are fired before less recently activated rules of the same salience. When the breadth strategy is active, rules of the same salience fire in the order in which they are activated. Note that in either case, if several rules are activated simultaneously (i.e., by the same fact-assertion event) the order in which these several rules fire is unspecified, implementation-dependent and subject to change. More built-in strategies may be added in the future. You can (perhaps) implement your own strategies in Java by creating a class that implements the `jess.Strategy` interface and then specifying its fully-qualified classname as the argument to `set-strategy`. Details can be gleaned from the source. At this time, though, I think some of the methods you'd need to call are package-protected.

8.151. (show-deffacts)

Package:
Scriptlib
Arguments:
None
Returns:
nil
Description:
Displays all defined deffacts to the 't' router.

8.152. (show-deftemplates)

Package:
Scriptlib
Arguments:
None
Returns:
nil
Description:
Displays all defined `deftemplate`s to the 't' router.

8.153. (show-jess-listeners)

Package:
Scriptlib
Arguments:
None
Returns:
nil
Description:
Displays all `JessListener`s registered with the engine to the 't' router.

8.154. (socket <Internet-hostname> <TCP-port-number> <router-identifier>)

Package:
MiscFunctions
Arguments:
An Internet hostname, a TCP port number, and a router identifier
Returns:
The router identifier
Description:
Somewhat equivalent to `open`, except that instead of opening a file, opens an unbuffered TCP network connection to the named host at the named port, and installs it as a pair of read and write routers under the given name.

8.155. (sqrt <numeric-expression>)

Package:
MathFunctions
Arguments:
A numeric expression
Returns:
Number
Description:
Returns the square root of its only argument.

8.156. (store <string or atom> <expression>)

Package:
Intrinsics
Arguments:
A string or atom and any other value
Returns:
(varies)
Description:
Associates the expression with the name given by the first argument, such that later calls to the `fetch` will retrieve it. Storing the atom nil will clear any value associated with name. Analagous to the `store()` member function of the `jess.Rete` class. See section on using store and fetch for

more details.

8.157. (str-cat <expression>*)

Package:
StringFunctions
Arguments:
Zero or more expressions
Returns:
String
Description:
Concatenates its arguments as strings to form a single string. For arguments of type RU.EXTERNAL_ADDRESS, the toString() method of the contained object is called.

8.158. (str-compare <string-expression> <string-expression>)

Package:
StringFunctions
Arguments:
Two atoms or strings
Returns:
Integer
Description:
Lexicographically compares two strings. Returns 0 if the strings are identical, a negative integer if the first is lexicographically less than the second, a positive integer if lexicographically greater.

8.159. (str-index <lexeme-expression> <lexeme-expression>)

Package:
StringFunctions
Arguments:
Two atoms or strings
Returns:
Integer or FALSE
Description:
Returns the position of the first argument within the second argument. This is the 1-based index at which the first string first appears in the second; otherwise, returns FALSE.

8.160. (stringp <expression>)

Package:
PredFunctions
Arguments:
One expression
Returns:
Boolean

Description:
Returns TRUE for strings; otherwise, returns FALSE.

8.161. (str-length <lexeme-expression>)

Package:
StringFunctions
Arguments:
An atom or string
Returns:
Integer
Description:
Returns the length of an atom in characters.

8.162. (subseq$ <multifield-expression> <begin-integer-expression> <end-integer-expression>)

Package:
MultiFunctions
Arguments:
A multifield and two numeric expressions
Returns:
Multifield
Description:
Extracts the specified range from a multifield value consisting of the elements between the two 1-based indices of the given multifield, inclusive.

8.163. (subsetp <multifield-expression> <multifield-expression>)

Package:
MultiFunctions
Arguments:
Two multifields
Returns:
Boolean
Description:
Returns TRUE if the first argument is a subset of the second (i.e., all the elements of the first multifield appear in the second multifield); otherwise, returns FALSE.

8.164. (sub-string <begin-integer-expression> <end-integer-expression> <string-expression>)

Package:
StringFunctions
Arguments:
Two numbers and a string

Returns:
String
Description:
Retrieves a subportion from a string. Returns the string consisting of the characters between the two 1-based indices of the given string, inclusive.

8.165. (symbolp <expression>)

Package:
PredFunctions
Arguments:
One expression
Returns:
Boolean
Description:
Returns TRUE for symbols; otherwise, returns FALSE.

8.166. (sym-cat <expression>*)

Package:
Intrinsics
Arguments:
Zero or more expressions
Returns:
Atom
Description:
Concatenates its arguments as strings to form a single symbol. For arguments of type RU.EXTERNAL_ADDRESS, the toString() method of the contained object is called.

8.167. (synchronized <java-object> <action>*)

Package:
MiscFunctions
Arguments:
Any Java object, followed by any number of expressions
Returns:
(varies)
Description:
Executes the expressions inside a Java "synchronized" block which locks the given object. Returns the value of the last expression evaluated.

8.168. (system <lexeme-expression>+ [&])

Package:
MiscFunctions

Arguments:
One or more atoms or strings
Returns:
a java.lang.Process object, or FALSE
Description:
Sends a command to the operating system. Each atom or string becomes one element of the argument array in a call to the Java `java.lang.Runtime.exec(String[] cmdaray)` method; therefore to execute the command `edit myfile.txt`, you should call `(system edit myfile.txt)`, not (system "edit myfile.txt").

Normally blocks (i.e., Jess stops until the launched application returns), but if the last argument is an ampersand (`&`), the program will run in the background. The standard output and standard error streams of the process are connected to the 't' router, but the input of the process is not connected to the terminal.

Returns the Java Process object. You can call waitFor and then exitValue to get the exit status of the process.

8.169. (throw <java-object>)

Package:
Intrinsics
Arguments:
A Java object that must inherit from java.lang.Throwable
Returns:
Does not return
Description:
Throws the given exception object. If the object is a `JessException`, throws it directly. If the object is some other type of exception, it is wrapped in a `JessException` before throwing. The object's stack trace is filled in such that the exception will appear to have been created by the `throw` function.

8.170. (time)

Package:
MiscFunctions
Arguments:
None
Returns:
Number
Description:
Returns the number of seconds since 12:00 AM, Jan 1, 1970.

8.171. (try <expression>* [catch <expression>*] [finally <expression>*])

Package:
Intrinsics
Arguments:
One or more expressions, followed optionally by the atom catch followed by zero or more expressions, followed optionally by the atom finally followed by zero or more expressions. Either the catch, or the finally, or both must be included.
Returns:
(Varies)
Description:
This command works something like Java `try` with a few simplifications. The biggest difference is that the `catch` clause can specify neither a type of exception nor a variable to receive the exception object. All exceptions occurring in a `try` block are routed to the single `catch` block. The variable `?ERROR` is made to point to the exception object as an `EXTERNAL_ADDRESS`. For example:

```
(try
(open NoSuchFile.txt r)
catch
(printout t (call ?ERROR toString) crlf))
```

prints

```
Rete Exception in routine _open::call.
Message: I/O Exception java.io.FileNotFoundException: NoSuchFile.txt.
```

An empty `catch` block is fine. It just signifies ignoring possible errors.

The code in the `finally` block, if present, is executed after all try and/or catch code has executed, immediately before the `try` function returns.

8.172. (undefadvice (<function-name> | ALL | <multifield>))

Package:
Intrinsics
Arguments:
A function name, or ALL, or a multifield of function names
Returns:
TRUE
Description:
Removes all advice from the named function(s).

8.173. (undefinstance (<java-object> | *))

Package:
ReflectFunctions
Arguments:
A Javaobject, or the atom *

Returns:
TRUE
Description:
If the object currently has a shadow fact, it is removed from the knowledge base. Furthermore, if the object has a `PropertyChangeListener` installed, this is removed as well. If the argument is "*" this is done for all `definstance`s.

8.174. (undefrule <rule-name>)

Package:
Intrinsics
Arguments:
An atom representing the name of a rule
Returns:
Boolean
Description:
Deletes a rule. Removes the named rule from the Rete network and returns `TRUE` if the rule existed. This rule will never fire again.

8.175. (union$ [<multifield-expression>]+)

Package:
MultiFunctions
Arguments:
One or more multifields
Returns:
Multifield
Description:
Returns a new multifield consisting of the union of all of its multifield arguments (i.e., of all the elements that appear in any of the arguments with duplicates removed).

8.176. (unwatch <watch-item>)

Package:
Intrinsics
Arguments:
One of the atoms all, rules, compilations, activations, facts, focus
Returns:
Description:
Causes trace output to not be printed for the given indicator. See `watch`.

8.177. (upcase <lexeme-expression>)

Package:
StringFunctions

Arguments:
A string or atom
Returns:
A string
Description:
Converts lowercase characters in a string or symbol to uppercase. Returns the argument as an all-uppercase string.

8.178. (update <java-object>+)

Package:
ReflectFunctions
Arguments:
One or more Java objects, previously passed as arguments to `definstance`.
Returns:
The fact-id of the shadow fact tied to the last argument.
Description:
Static definstances aren't updated automatically, since Jess doesn't know when a definstanced object has been changed. This function lets you tell Jess explicitly that one or more Java objects have been updated. In response, Jess will find their corresponding shadow facts and update all their slots.

8.179. (view)

Package:
ViewFunctions
Arguments:
None
Returns:
TRUE
Description:
This Userfunction is included in the Jess distribution but is not normally installed. You must load it using `load-package` (the class name is `jess.ViewFunctions`). When invoked, it displays a live snapshot of the Rete network in a graphical window. See How Jess Works for details.

8.180. (watch (all | rules | compilations | activations | facts))

Package:
Intrinsics
Arguments:
One of the atoms all, rules, compilations, activations, facts, focus activations
Returns:
TRUE
Description:
Produces additional debug output when specific events happen in Jess, depending on the argument. Any number of different watches can be active simultaneously:

- `rules`: prints a message when any rule fires.
- `compilations`: prints a message when any rule is compiled.

- `activations`: prints a message when any rule is activated, or deactivated, showing which facts have caused the event.
- `facts`: print a message whenever a fact is asserted or retracted.
- `focus`: print a message for each change to the module focus stack.
- `all`: all of the above.

8.181. (while <expression> [do] <action>*)

Package:
    Intrinsics
Arguments:
    A Boolean value or a function call returning Boolean, the atom do, and zero or more expressions
Returns:
    (Varies)
Description:
    Allows conditional looping. Evaluates the boolean expression repeatedly. As long as it does not equal `FALSE`, the list of other expressions are evaluated. The value of the last expression evaluated is the return value.

Back to index

10. The Rete Algorithm

The information in this Section is provided for the curious reader. An understanding of the Rete algorithm may be helpful in planning expert systems; an understanding of Jess's implementation probably will not. Feel free to skip this section and come back to it some other time. You should not take advantage of many of the Java classes mentioned in this section. They are internal implementation details and any Java code you write which uses them may well break each time a new version of Jess is released.

Jess is a rule-based expert system shell. In the simplest terms, this means that Jess's purpose it to continuously apply a set of if-then statements (*rules*) to a set of data (the *knowledge base*). You define the rules that make up your own particular expert system. Jess rules look something like this:

```
(defrule library-rule-1
  (book (name ?X) (status late) (borrower ?Y))
  (borrower (name ?Y) (address ?Z))
 =>
  (send-late-notice ?X ?Y ?Z))
```

Note that this syntax is identical to the syntax used by CLIPS. This rule might be translated into pseudo-English as follows:

```
Library rule #1:
If
  a late book exists, with name X, borrowed by someone named Y
and
  that borrower's address is known to be Z
then
  send a late notice to Y at Z about the book X.
```

The book and borrower entities would be found on the knowledge base. The knowledge base is therefore a kind of database of bits of factual knowledge about the world. The attributes (called *slots*) that things like books and borrowers are allowed to have are defined in statements called `deftemplates`. Actions like `send-late-notice` can be defined in user-written functions in the Jess language (`deffunctions`) or in Java (`Userfunctions`) For more information about rule syntax refer to the the Jess language guide.

The typical expert system has a fixed set of rules while the knowledge base changes continuously. However, it is an empirical fact that, in most expert systems, much of the knowledge base is also fairly fixed from one rule operation to the next. Athough new facts arrive and old ones are removed at all times, the percentage of facts that change per unit time is generally fairly small. For this reason, the obvious implementation for the expert system shell is very inefficient. This obvious implementation would be to keep a list of the rules and continuously cycle through the list, checking each one's left-hand-side (LHS) against the knowledge base and executing the right-hand-side (RHS) of any rules that apply. This is inefficient because most of the tests made on each cycle will have the same results as on the previous iteration. However, since the knowledge base is stable, most of the tests will be repeated. You might call this the *rules finding facts* approach and its computational complexity is of the order of O(RF^P), where R is the number of rules, P is the average number of patterns per rule LHS, and F is the number of facts on the knowledge base. This escalates dramatically as the number of patterns per rule increases.

Jess instead uses a very efficient method known as the Rete (Latin for *net*) algorithm. The classic paper on the Rete algorithm *("Rete: A Fast Algorithm for the Many Pattern/ Many Object Pattern Match Problem", Charles L. Forgy, Artificial Intelligence 19 (1982), 17-37)* became the basis for a whole generation of fast expert system shells: OPS5, its descendant ART, and CLIPS. In the Rete algorithm, the inefficiency described above is alleviated (conceptually) by remembering past test results across iterations of the rule loop. Only new facts are tested against any rule LHSs. Additionally, as will be described below, new facts are tested against only the rule LHSs to which they are most likely to be relevant. As a result, the computational complexity per iteration drops to something more like O(RFP), or linear in the size of the fact base. Our discussion of the Rete algorithm is necessarily brief. The interested reader is referred to the Forgy paper or to *Giarratano and Riley, "Expert Systems: Principles and Programming", Second Edition, PWS Publishing (Boston, 1993)* for a more detailed treatment.

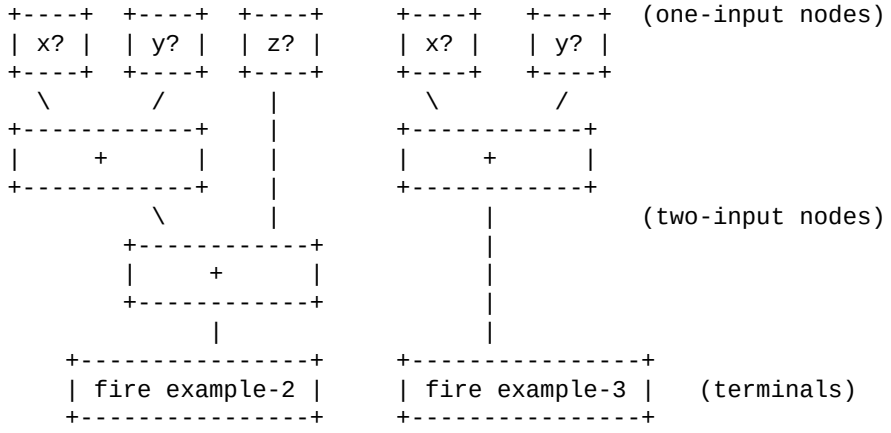
The Rete algorithm is implemented by building a network of nodes, each of which represents one or more tests found on a rule LHS. Facts that are being added to or removed from the knowledge base are processed by this network of nodes. At the bottom of the network are nodes representing individual rules. When a set of facts filters all the way down to the bottom of the network, it has passed all the tests on the LHS of a particular rule and this set becomes an *activation*. The associated rule may have its RHS executed (*fired*) if the activation is not invalidated first by the removal of one or more facts from its activation set.

Within the network itself there are broadly two kinds of nodes: one-input and two-input nodes. One-input nodes perform tests on individual facts, while two-input nodes perform tests across facts and perform the grouping function. Subtypes of these two classes of node are also used and there are also auxilliary types such as the terminal nodes mentioned above.

An example is often useful at this point. The following rules:

```
(defrule example-2      (defrule example-3
   (x)                     (x)
   (y)                     (y)
   (z)                     => )
   => )
```

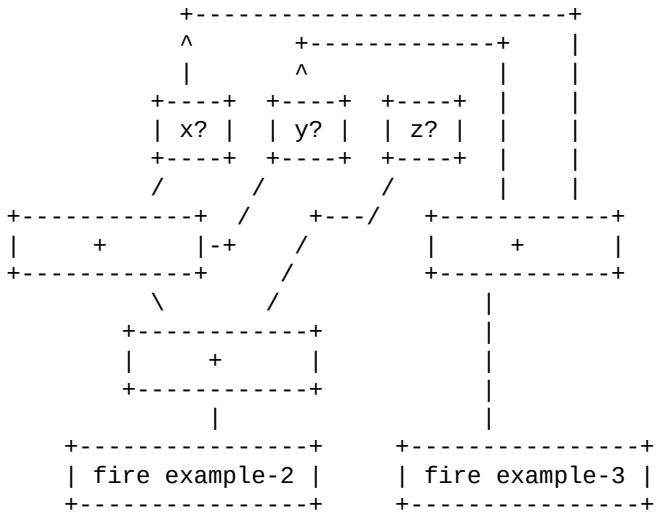
might be compiled into the following network:



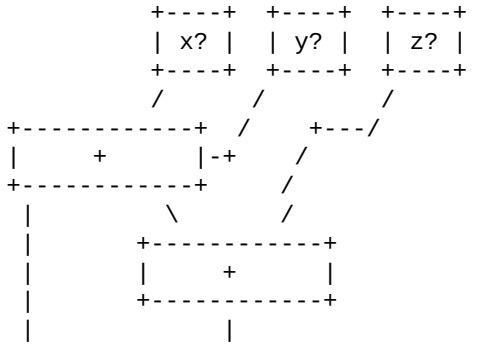
The nodes marked `x?`, etc., test if a fact contains the given data, while the nodes marked `+` remember all

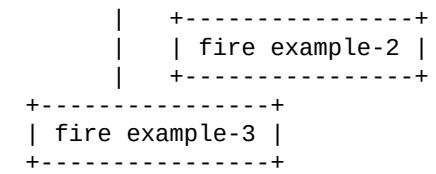
facts and fire whenever they've received data from both their left and right inputs. To run the network, Jess presents new facts to each node at the top of the network as they added to the knowledge base. Each node takes input from the top and sends its output downwards. A single input node generally receives a fact from above, applies a test to it, and, if the test passes, sends the fact downward to the next node. If the test fails, the one-input nodes simply do nothing. The two-input nodes have to integrate facts from their left and right inputs, and in support of this, their behavior must be more complex. First, note that any facts that reach the top of a two-input node could potentially contribute to an activation: they pass all tests that can be applied to single facts. The two input nodes therefore must remember all facts that are presented to them, and attempt to group facts arriving on their left inputs with facts arriving on their right inputs to make up complete activation sets. A two-input node therefore has a *left memory* and a *right memory*. It is here in these memories that the inefficiency described above is avoided. A convenient distinction is to divide the network into two logical components: the single-input nodes comprise the *pattern network*, while the two-input nodes make up the *join network*.

There are two simple optimizations that can make Rete even better, The first is to share nodes in the pattern network. In the network above, there are five nodes across the top, although only three are distinct. We can modify the network to share these nodes across the two rules (the arrows coming out of the top of the `x?` and `y?` nodes are outputs):



But that's not all the redundancy in the original network. Now we see that there is one join node that is performing exactly the same function (integrating x,y pairs) in both rules, and we can share that also:





The pattern and join networks are collectively only half the size they were originally. This kind of sharing comes up very frequently in real systems and is a significant performance booster!

You can see the amount of sharing in a Jess network by using the `watch compilations` command. When a rule is compiled and this command has been previously executed, Jess prints a string of characters something like this, which is the actual output from compiling rule example-2, above:

```
example-2: +1+1+1+1+1+1+2+2+t
```

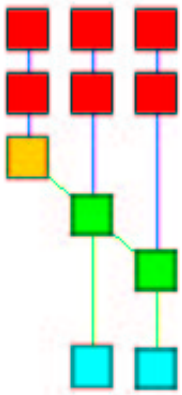
Each time `+1` appears in this string, a new one-input node is created. `+2` indicates a new two-input node. Now watch what happens when we compile example-3:

```
example-3: =1=1=1=1=2+t
```

Here we see that `=1` is printed whenever a pre-existing one-input node is shared; `=2` is printed when a two-input node is shared. `+t` represents the terminal nodes being created. (Note that the number of single-input nodes is larger than expected. Jess creates separate nodes that test for the head of each pattern and its length, rather than doing both of these tests in one node, as we implicitly do in our graphical example.) No new nodes are created for rule example-3. Jess shares existing nodes very efficiently in this case.

Jess's Rete implementation is very literal. Different types of network nodes are represented by various subclasses of the Java class `jess.Node`: `Node1`, `Node2`, `NodeNot2`, `NodeJoin`, and `NodeTerm`. The `Node1` class is further specialized because it contains a *command* member which causes it to act differently depending on the tests or functions it needs to perform. For example, there are specializations of `Node1` which test the first field (called the *head*) of a fact, test the number of fields of a fact, test single slots within a fact, and compare two slots within a fact. There are further variations which participate in the handling of multifields and multislots. The Jess language code is parsed by the class `jess.Jesp`, while the actual network is assembled by code in the class `jess.ReteCompiler`. The execution of the network is handled by the class `Rete`. The `jess.Main` class itself is really just a small demonstration driver for the jess package, in which all of the interesting work is done.

The `view` command is a graphical viewer for the Rete network itself; I have used this as a debugging tool for Jess, but it may have educational value for others, and it may help you to design more efficient systems of rules in Jess. Issuing the `view` command after entering the rules example-2 and example-3 produces a very good facsimile of the drawing (although it correctly shows the larger number of one-input nodes). The various nodes are color-coded according to their roles in the network; `Node1` nodes are red; `Node2` nodes are green; `NodeNot2` nodes are yellow; and `Defrule` nodes are blue. The orange node in the figure is a "right-to-left adapter" node; one of these is always used to connect the first pattern on a rule's LHS to the network. Passing the mouse over a node displays information about the node and the tests it contains; double-clicking on a node brings up a dialog box containing the same information (for join nodes, the memory contents are also displayed, while for `Defrule` nodes, a pretty-print representation of the rule is shown). See the description of the `view` function for important information before using it.



Back to index

11. Change History

Version 6.1RC1 (March 24th, 2003):

Fixed another leak and another deadlock in "logical" (thanks Glenn Williams.) Reflection machinery now tries more aggressively to convert between floats and RU.FLOATs, ints and RU.INTEGERs, etc, in both directions. assert-string just returns FALSE on duplicate fact (thanks Thomas Diesler.) Better system for renaming variables in nested NOTs. Fix a race condition in JessEventSupport (thanks Chad Loder.)

Version 6.1b3 (February 28th, 2003):

Most of the Rete "listXXX" methods lock and copy to prevent ConcurrentModificationExceptions. Fix problem with triply nested (not) (thanks Alan Moore.) Fix a race condition with modify and run-until-halt (Mr. Moore, again.) "unique" CE is now a no-op. Deadlocks, memory leaks, and other issues with the "logical" CE repaired (many thanks to Travis Nelson and Glenn Williams.)

Version 6.1b2 (February 14th, 2003):

Proper handling for "OR" CE's deeply nested inside NOTs. Nested, negated "test" CEs now work properly. Add "getThisActivation" and "getThisRuleName" to Rete (thanks Richard Kasperowski.) Can test fact bindings against previously defined variables. Simple public ConditionalElement class. "exists" handled by parser.

Version 6.1b1 (January 28th, 2003):

Code cleanup: remove unused return value from callNodeLeft/Right, add Context argument to callNodeLeft/Right, and remove ThreadLocal Rete.s_engines and static Rete.getEngine() method. When any fact, shadow or not, is modified, either from Jess, from the Java API, or from a Bean event, a single FACT | MODIFIED event will be generated. Many of the built-in package classes are no longer public. Added Rete.updateObject and (update). Added Rete.modify(). Added Rete.watch(), unwatch(). (watch facts) reports modify again. max-background-rules declaration added to defquery. Javadoc coverage improved. Fixed ppdeftemplate handling of default-dynamic (thanks Ross Judson.)

Version 6.1a5 (January 15th, 2003):

Fixed a bug with direct matching of fact bindings (thanks Morten Vigel). Fixed a typo in Node1TNEV1 (Ralph Grove.) Added "synchronized" and "get-strategy" functions. Calling "set-strategy" now changes the default strategy for new modules. Fixed a deadlock in defquery (thanks Chad Loder.) Public interface "Test" renamed to "TestBase." Fixed bug in pretty-printing of deftemplates. Performance is back up to 6.0 equivalent.

Version 6.1a4 (August 30th, 2002):

Fixed a 6.1a3 bug where negated patterns could trigger backward chaining. "eval" and "build" can use variables from the calling context. Clarify and extend "app object" mechanism; Jess can be installed as a standard extension. Partially ameliorate performance regression from 6.1a3.

Version 6.1a3 (July 23rd, 2002):

Fixed a bug in save-facts (thanks Scott Trackman). Fixed deadlock problem in retract/undefinstance (thanks Blaine Bell.) Mihai Barbuceanu helped modernize the "bag" command. Various API enhancements. New Rete constructor with "app-object" argument. Massive refactoring of rule compiler to handle arbitrarily nested AND and NOT CEs in a new, simpler way; universal quantifiers are now handled correctly. Fixed a bug with the logical CE when adding new rules to a running system. Undefinstanced, nonserializable Beans now no longer prevent Rete serialization (thanks James Gallogly and Alan Moore.)

Version 6.1a2 (May 22st, 2002):

Fixed early return bug in try/catch/finally (thanks Chad Loder). (modify) now immediately modifies shadow fact for static definstances. A shadow fact can tell you the category of definstance it belongs to. New version of `format` function now has a %n conversion specifier, and also fixed an infinite loop when formatting floating-point zeroes. Refactoring in logical -- should not be user visible.

Version 6.1a1 (April 3rd, 2002):

Class import table now public. "view" cmmand improved, and bugs fixed. Activations now have individual salience values. Many methods in jess.Context now public. Rete.retract() now calls undefinstance() if needed. Rete.undefinstance() returns shadow fact. Matching against defglobal works (thanks Simon Hamilton.) (logical) bug fixed (thanks Blaine Bell.) Fixed a bug executing tests in Node2 (thanks Ed Katz.) Fixed a bug with inadvertent changes to current module (thanks John Callahan.) Fixed an (exists) bug and an '|'-constraint bug (thanks Jack Kerkhof).

Version 6.0 (December 7th, 2001):

defclass and definstance now accept defclass names qualified wih module names (thanks Juraj Frivolt.) ppdefrule and friends don't throw if the argument is bad (thanks Mikael Rundqvist.) Release notes and porting info added to docs. Other small patches.

Version 6.0b3 (November 2nd, 2001):

nth$ returns nil instead of throwing exceptions (thanks Seung Lee.) Deffunction has public API for fetching arguments and actions (thanks Scott Kaplan, Henrik Eriksson). If pattern-matches during (modify) throws an exception, the modified fact no longer disappears from the fact-list (thanks Mihai Barbuceanu.) If pattern-matches during addDefrule throw, rule is still added to rulebase (thanks Mikael Rundqvist). Fixed rounding bug in format function (thanks Mike Isenberg). Fixed issue with variable renaming when an 'and' CE was nested in a 'not' CE (thanks Drew Van Duren). Can omit the functor 'call' even if the Java object is the return value of a function call. retract, assert, and duplicate will once again accept an integer as their first argument -- the fact-id function is no longer necessary (although it can still be used.) You can put pattern bindings inside of grouping CEs. (undefinstance *) now works as advertised (thanks Jason Smith). Remove "modify-n" rewrite optimization, so pattern binding to (or) CEs works.

Version 6.0b2 (October 3rd, 2001):

Can redefine a defclass with identical definition (thanks Ian de Beer). Single-stepping rule execution was losing activations (thanks Simon Blackwell.) Fix a bug preventing FuzzyJess from working with non-Fuzzy multifields (thanks Bob Orchard.) explode$ can deal with embedded comments (thanks Simon Blackwell.) (logical) can handle nested (not) and (exists).

Version 6.0b1 (September 13th, 2001):

Fixed a bug in compiling rules with function calls inside of (not (and)) constructs (thanks Thomas Gentsch). Broadcast DEFINSTANCE and DEFCLASS events (ibid.) and RUN and HALT events. defmodules, focus stack, etc., implemented. (clear) doesn't delete Java Userfunctions (thanks Glenn Tarbox.) Rename SerializablePropertyDescriptor to SerializablePD so the class name isn't too long for the Mac filesystem (thanks Matt Bishop). (rules), (facts), others, now in Java, not scriptlib. Redefining a deftemplate throws an exception, unless the definitions are identical. Added "duplicate" function. sym-cat and str-cat call toString() on EXTERNAL_ADDRESS arguments (thanks Chad Loder.) Many other bug fixes and cleanups.

Version 6.0a8 (July 19th, 2001):

Docstrings for deftemplates properly parsed (thanks Simon Hamilton.) Several features broken in binary-only distribution (view, ppdefrule, etc) are now fixed. JessEvents of type USERFUNCTION_CALLED now include the Userfunction as the user object, not the FunctionHolder (thanks Chad Loder.)

Version 6.0a7 (June 1st, 2001):

New pretty-print architecture which shows how things are actually represented; based on new Visitor interface. ppdefX functions now return, rather than print, their results. show-jess-listeners works again. Simplify how negated patterns are parsed. JAR file not an executable JAR anymore. Fix Thread problem with runUntilHalt, deadlock in propertyChange (thanks Charles May.) deffunctions accepting only a wildcard can now be called with no arguments (thanks Thomas Gentsch.) No more erroneous undefined variable message (Gentsch.) The minus (-) function works if one or more arguments are atoms, just as the other binary math functions do (thanks Pau Ortega.) The first arg to (modify) can be a funcall. (return) from (try) or (catch) block no longer skips second and subsequent expressions in (finally) block (thanks Chad Loder.) Deadlock in one form of Java assert() (Loder.) (or), (and), and (not) CEs can be nested in structures of arbitrary complexity.

Version 6.0a6 (May 3rd, 2001):

There are no more "optional" functions. Fact.getTime() is public (thanks Alan Moore.) Lots of small improvements to manual. Fix a bug in pattern-matching fact-ids (thanks Matthew Johnson.) Slightly different rules for loading batch files from jars; now they must omit the package-dir name. The "logical" CE is supported.

Version 6.0a5 (March 12th, 2001):

Fixed a serious bug in "modify" (thanks Bob Orchard.) Some documentation cleanup. runQuery method returns Iterator.

Version 6.0a4 (March 8th, 2001):

"Watch" listeners were not being cleared. Use a priority queue for agenda -- modest performance improvement for most applications, big improvement for conflict-resolution-limited apps. Strategy interface vastly simplified, rewrote built-in strategies. FactList.ppFacts(), and hence (save-facts), rewritten to write directly to a Writer, greatly decreasing memory usage. Quit button removed from Console (thanks Andrew Marshall.) Calling a native function that returns a wrapper type (Integer, Float, etc) now gives an EXTERNAL_ADDRESS holding the wrapper object, instead of unwrapping the data. Incompatible changes to improve FuzzyJess performance and fix possible bug; a new version of FuzzyJess will be required to work with Jess 6.0a4. Fact-duplication concept eliminated -- fact duplication is never allowed. Fact list stored on a HashMap, not a Vector; Now finding is fast, enumerating is slow. For the time being, Jess 6.0XX now requires Java 2.

Version 6.0a3 (January 25th, 2001):

Bload and bsave now work much better, using real, complete serialization. Serializing/deserializing Rete saves and restores everything except the I/O router tables. Rete loads scriptlib itself, so you don’t need to do it yourself anymore. Lots of refactoring -- Rete now exports more useful functions, but delegates their implementations; for example, there are public defclass(), definstance(), and undefinstance() methods. A small bug in Tokenizer fixed (thanks Norman Ghyra.) Methods added to support Thomas Barnekow’s JessX Jess extensions.

Version 6.0a2 (July 20th, 2000):

Defqueries can now backwards-chain to get results. run-until-halt now clears halt flag first. finally in (try) (thanks Thomas Barnekow). Many typos in manual fixed (thanks Michael Futtersack.) (or) and (and) CEs, and general nesting of CEs now operational (thanks to Mariusz Nowostawski and Jack Kerkhof for test cases,) although code still needs refactoring and (and) and (or) won’t work correctly inside of (not). waitForActivations() could block even if activations were available; now fixed (thanks Thomas Barnekow). Added some "LISP compatibility functions:" progn, apply. Fixed off-by-one error in line-numbers in error reports. Vastly improved efficiency of multifield matching (thanks to Sebastian Varges for noticing a problem with the old implementation.) Deactivate activations earlier in the rule-firing process, to remove a bit of redundant processing (thanks David

Bruce.).

Version 6.0a1 (April 25th, 2000):

Fact-id's are now simply references to Fact objects. Use the supplied fact-id function to create a fact-id value from an integer. Many memory-usage reductions; no more extra Vectors in Rete network classes. Several code-shrinking refactorizations. Rete class no longer delivers events to listeners registered by a handler for the same event (thanks Alex Karasulu.) Fact-ids no longer change on modify. (batch) now works on a .clp files in a .jar file from an applet (thanks Javier Torres).

Version 5.2 (June 30th, 2000):

Bug fix release. waitForActivations() could block even if activations were available; now fixed (thanks Thomas Barnekow).

Version 5.1 (April 24th, 2000):

Bug fix release. Fixed two bugs in backwards chaining (thanks Ashraf Afifi.) Also, a regression repaired: pattern bindings couldn't be matched as they could in Jess 4, and now they can again.

Version 5.0 (January 28th, 2000):

Added new function cross-reference section to the manual (thanks to Roger Firestone for the idea.) parseDeftemplate now handles comments in child templates (thanks Richard Long.) TextAreaWriter (and hence Console and ConsoleApplet) prunes the oldest characters from its TextArea to prevent the Win32 freeze problem.

Version 5.0b4 (January 5th, 2000):

Fixed broken fact-slot-value function. Some support for new RU.LONG type. Added (run-until-halt), runUntilHalt(), activationSemaphore, etc. (unique) CE no longer prevents partial matches from being retracted, only from being propagated (thanks Vicken Kasparian.) Nested nots and (exists) CE added. Unmatched variables can appear in function calls within defqueries (thanks Michal Fadljevic). Bsave/bload now works with defclasses (thanks Russ Milliken.) ReflectFunctions cache result of BeanInfo.getPropertyDescriptors().

Version 5.0b3 (November 30th, 1999):

Support for reworked version of Bob Orchard's fuzzy logic extensions. When multiple activations of one rule were due to different multifield matches of the same set of facts, retracting one fact would cause only one activation to be removed. This is now fixed (thanks, Al Davis.) Share function-call-containing join nodes; thanks George Rudolph for the reminder. Fixed UnWatch class for buggy JDK 1.x compilers. Can match directly on defglobals now. ReflectFunctions cache result of Class.getMethods().

Version 5.0b2 (November 8th, 1999):

Several bugs in run-query and parseDefquery fixed (thanks Alex Karasulu). (watch) handlers flush stdout (thanks David Young.) Fix bare return-value constraints in multislots (thanks Dave Barnett.) Problem with returning parameters from deffunctions, and missing resolveValue calls in BagFunctions fixed (thanks Bruce Douglas.) assert() and retract() call processPendingFacts() directly (thanks Thomas Barnekow.) Multislot matching "optimization" fixed (thanks Robert Gaimari.) Deffacts allows global variables in fact slots (thanks Michael Coen.) Try harder to coerce all multifield entries to same type when converting to Java array (thanks Ralph Grove.) Fix a synchronization problem (thanks Javier Maria Torres Ramon.)

Version 5.0b1 (September 21st, 1999):

jess.Console works again (thanks Lakshmi Vempati). Compiles with buggy JDK 1.1.x compiler. bload/bsave preserves listeners (i.e., deffacts work). Did away with Successor class (simplified Rete network, reduced memory usage.) Fixed bug where redefined rules with shared nodes wouldn't reset. Specialized Value subclasses now explicitly override numericValue(). Yet another pesky "not" bug (thanks Vadim Zaliva.) All facts/definstances inherit, ultimately, from "__fact". In Rete network,

callNodeRight() takes Fact, not token; right memories store Facts, too (fewer allocations, less memory!) Return-value constraint was erroneously requiring bound variable; thanks Dave Kirby. Miroslav Madecki reported a "ghost fact" bug; fixing it improved Jess's "manners" benchmark performace by another 25% (!) "system" returns a Process object (thanks Alan Moore). (open) uses a 10-character buffer - large file I/O speedup (thanks Norman Gyhra).

Version 5.0a6 (July 8th, 1999):

insert$, replace$ behave as documented w.r.t argument types (thanks Emmanuel Pierre). Typo: agenda wasn't being cleared on reset (thanks Bob Orchard.) Removed some redundant checks in network (more speedups!) Final multivar argument to deffunction now properly handles passed-in multifields; thanks Ning Zhong. Added "import". Fixed bug in using variable as salience value; thanks David Young. Bug in special code for calling public methods of package-private classes; thanks Cheruku Srini. Bug in member$ fixed (thanks David Young again.) Bug in retracting facts which are multifield pattern-matched on rule LHS fixed (thanks Yang Xiao). All classes serializable again (thanks Michael Friedrich). Several small optimizations related to fact-duplication (thanks Osvaldo Pinali Doederlein.) JessEvents working again, with event mask. Bitwise arithmetic functions added. Fixed some more multifield matching strangeness (thanks Benjamin Good). Added "|" (or) connective constraint.

Version 5.0a5 (May 20th, 1999):

Drastic changes to the way Funcalls are executed (much simplification and speedup!) Fixed "corrupted negcnt" when retracting definstance facts (thanks Abel Martinez.) Much improved agenda management, including time tags. Smart indexing based on tested values. New heuristics for eliminating redundant tests. (store foo nil) now removes value of foo (thanks Kathy Lee Simunich for the suggestion.) (system) command now pipes program output to console (thanks Fang Liu for idea). (undefinstance *) now removes all definstances (thanks Mike Lucero for idea). Multiple defglobal references on rule LHS no longer broken (thanks Jacek Gwizdka for report.) Many upgrades to JessException class (thanks Kenny Macleod). Some Value constructors no longer have the redundant second argument. get/set-fact-duplication (thanks to Osvaldo Pinali Doederlein for the idea.) Bob Orchard figured out how to get rid of the extra EOFs in jess.ConsolePanel. Console and ConsoleApplet can now read a file and continue, as they used to. Backwards chaining now more general; one goal can trigger multiple rules, both in parallel and in chains. ViewFunctions and ReflectFunctions moved into jess package, many more classes now package-private. jess.reflect package renamed jess.awt; TextReader, TextAreaWriter moved into it. "throw" command added (thanks Eric Eslinger).

Version 5.0a4 (March 18th, 1999):

Call removePropertyChangeListener on undefinstance (thanks to Dan Larner.) Added defadvice. Three buglets, thanks to Bob Orchard: defglobals can now refer to other Defglobals in their definitions; (bind) and (foreach) are now pickier about arguments; undefined variables now universally evaluate as 'nil'. Applet security problems fixed (thanks S.S. Ozsariyildiz). Problem with matching and otherwise dealing with zero-length multislots and array bean properties fixed; null array values are converted into zero-length arrays (thanks Miroslav Madecki.) (call) does much better error reporting. Passing Strings to functions accepting Objects now works (J. P. van Werkhoven.)

Version 5.0a3 (February 12th, 1999):

Token(Token) changes UPDATE to ADD. Hashcode used in TokenTree modifiable. (clear) now fully clears deffacts. Many general speedups. TokenTree hash factor can now be set globally and per-rule. Functional Accelerator class exists for math functions. (modify) funcalls print out correctly. get-var removed (no longer needed.) Nonsychronization error in jess.Retract.call() led to Corrupted Negcnt; fixed (reported by Dan Larner). Other corrupted negcnt error due to duplicated tokens fixed.

definstances can be static or dynamic. jess.Pattern is public; several methods added or made public to support rule editors. Defrule.toString works for all cases. Console class now uses Main to do business. .jessrc file now named scriptlib.clp and found using Class.getResource(). (batch) can use getResource() to find scripts, so they can live in .jar files. Batch now an intrinsic. Rewrote some multifield functions to better handle EXTERNAL_ADDRESS objects; added hashCode() to jess.Value. (facts) and (rules) now in scriptlib, not Java. Much improved public interface for Deftemplate, Fact classes.

Version 5.0a2 (December 15th, 1998):

m_sortcode in Token no longer a blank final to work around JDK 1.1.6/7a javac bug. Makefile no longer mentions unbundled files. Backward chaining works again (was broken in 5.0a1)

Version 5.0a1 (December 9th, 1998):

All 16-bit (Reader/Writer) text I/O - some user code changes required. Serialization (crude version). Inheritance for deftemplates, defclasses. jess.Main reads $HOME/.jessrc at startup if it exists. No more ReteDisplay, Resetable, or Clearable interfaces: instead, use jess.JessListener and jess.JessEvent. (view) uses JessListener instead of Observer/Observable; rewritten to provide movable nodes, per-node debug code. jess.reflect.JessListener class renamed to jess.reflect.JessAWTListener. Bean-itized many function names (name() -> getName()), breaking lots of user code, I'm afraid. (run) returns number of rules fired. Ordered facts now stored as unordered with one multislot named __data. Backwards chaining added. Bload and Bsave added.

Version 4.5 (April 15th, 1999):

Fixed "corrupted negcnt" when retracting definstance facts (thanks Abel Martinez.)

Version 4.4 (March 18th, 1999):

Token(Token) changes UPDATE to ADD. (clear) not truly clearing deffacts (thanks to Rob Jefson). Passing Strings to Java methods expecting Object now works. Call removePropertyChangeListener in undefinstance.

Version 4.3 (December 1st, 1998):

Fixed redundant default-value processing, which was leading to odd problems with definstances with null slot values (thanks to S.S. Ozsariyildiz). Removed intern()s from Tokenizer (faster compilation). Fixed NIL/nil ambiguity in ReflectFunctions (thanks Andreas Rasmussen.)

Version 4.2 (November 12th, 1998):

Fixed 'Corrupted negcnt' bug (thanks to Todd Bowers). (if ... then) function now throws an exception if atom 'then' is missing. Version string in 4.1 final was inadvertently left at 4.1b6. Added section to README explaining rule LHS semantics a bit better. Rete.findFactByID() is now public. Fix for very tricky 'phantom fact' problem reported by Steve Bucuvalas. Java method calls on Jess Strings now work for all Strings, not just alphanumeric ones. "animals" example modified to work with transitional gensym implementation.

Version 4.1 (September 15th, 1998):

Some minor bug fixes; code to allow you to leave off the '$' on a multivar after its first use, as in CLIPS.

Version 4.1b6:

Allow named variables in (not) CEs as long as they're not used in subsequent CEs. Fix a bug that was causing (return) to not work if inside a (foreach) inside a deffunction. Recursive deffunctions now work again. Jess works around a bug in some versions of Java that was preventing the atom '-' from parsing. Rete.listDefglobals() no longer lists duplicates of redefined defglobals (Karl Mueller found this one.) ReteDisplay.fireRule() is now called as appropriate. Accessing pattern-binding variables on rule LHSs works again (Karl again.) (reset) wasn't clearing all activations (thanks Al Davis); fixed. Funcall.toString() puts parens around the ValueVector version.

Version 4.1b5:

Just remove some debug code and extra files inadvertently shipped with 4.1b4.

Version 4.1b4:

addUserfunction, addDeffunction, etc collapsed into one addUserfunction routine in Rete class; same with findUserfunction. RU.getAtom() and RU.putAtom are gone! Userfunction.name() now returns String. ControlStructure interface used to clean up handling of such things. ReteCompiler uses Hashtables of Bindings instead of int[][] for vartables. Added default-dynamic deftemplate slot qualifier. Added set-/get-reset-globals, and changed the default defglobal reset behaviour. Added dynamic rule salience. Removed arbitrary limit of 32 slots for ordered facts and 32 tests per slot for all facts. "unique" CE (15-30% speed increase for many problems!) Various documentation improvements (many thanks to Win Carus.) Better error reporting (addContext() call in JessException.) Malformed calls to 'eval' or 'build' or 'executeCommand' no longer go into an infinite loop on EOF. Added "store" and "fetch". Added "external-addressp". Rearranged Test1, Test2 classes into an inheritance hierarchy with a virtual doTest method, allowing for alternate implementations (undocumented java-test functionality included). Value class will do more type conversions automatically. Final multifield argument of a deffunction now acts as a wildcard, as in CLIPS (thanks David Young.)

Version 4.1b3

Problem with calling public methods of package-private classes from Jess fixed thanks to Lars Rasmusson's explanation. OutOfMemoryError while parsing file containing unbalanced open parens fixed. Line breaks in double-quoted strings no longer need to be (but can be) escaped. Two fixes thanks to Andreas Rasmusson: gensym* returns a symbol as documented, not a string; and a propertyChangeEvent for a bogus property no longer causes Jess to retract a definstance without updating it. Many of the synchronized methods in the Rete class no longer are synchronized; instead they use either synchronized blocks keyed to affected members or simply depend on the internal synchronization of Hashtables. read and readline explicitly act differently for console-like and file-like streams. ConsoleDisplay gets a Clear Window button.

Version 4.1b2

Bug in character-escape lexing fixed thanks to Josiah Poon. Parser-related bug in explode$ fixed thanks to Andrew X. Comas. eval, build, executeCommand() again properly return the result of last expression, not EOF. min, max take arbitrary # of arguments. implode$ now works; it apparently never really did. printout puts parens around multifields again. str-compare documentation corrected. undefinstance now removes the fact representing an object as well as deactivating matching. Wrote large regression test suite (not included in distribution). Bug in multiple simultaneous Rete.run() calls in separate threads fixed thanks to Andreas Rasmusson. Selectable conflict resolution strategies (only depth and breadth supported now) and user-definable strategies. The try command is added.

Version 4.1b1

Much better lexer (no more StreamTokenizer). Input buffering problems with JDK 1.1.2-1.1.4 fixed. Bug in (test) CE fixed. Can call run on rule RHS. Bug in incremental update fixed. Separate command-line, applet, and GUI console driver classes (Quiz* classes broken up, renamed to Console*). read and readline should work exactly as in CLIPS. Manual describes more about how to write Java main(). Bug in definstance that was preventing use of subclasses of a defclassed class is fixed.

Version 4.0

BeanInfo support. quiz.html embeds only one QuizDisplay applet. Pumps demo works again (sorry). Conflict resolution strategy now should be exactly the same as CLIPS's default.

Version 4.0b4

Extensive manual rewrite, adding lots of Java/Jess interoperation info. Allow standard CLIPS deffunction docstrings. Thanks to Jack Fitch, Dave Carlson and Alex Jacobson, property names for reflected Java Beans now use standard capitalization transform. Better error reporting, especially during parsing and from the command line. set and get renamed to set-member and get-member. set and get are now functions that read and write Bean properties. ppdefrule properly handles quoted strings in function calls. executeCommand and friends reuse a single parser. Thanks to Karl Mueller for Rete.retractString. Taught batch to read applet-based data files. eval now handles non-sexps. Better mechanism for synchronizing streams. QuizDisplay is an applet as well as an application. run accepts an argument, the maximum number of rules to fire. Fixed bug in modify when new slot value was a zero-length multifield. Fixed ReteCompiler bug where MTELN nodes were not consistently generated for zero-length multifield matches. Thanks to Sidney Bailin, fixed problem with accessing defglobals and variables bound to pattern indexes on rule LHSs. Added get-var function. Added undefinstance. modify and retract now handle definstance facts specially. Fixed some doPPPattern bugs (Dave Carlson again!).

Version 4.0b3

Added jess.reflect package containing new, call, set, and get. Added JessListener and its subclasses. added engine. Changed printing of external-addresses to include Java class name. Changed parser to accept variable names as Funcall heads (call is substitued, resulting in a runtime error if call is not installed). and and or functions now accept any values as arguments, not only funcalls. Added foreach control structure. Command prompt doesn't print NIL return values. Fixed another not bug (thanks to Sidney Bailin). Added matching of Java objects on rule LHSs: definstance, defclass. TokenTree now uses sortcode % 101 as hash key, not the sortcode itself. All global classes moved into jess package. Jess class renamed Main.

Version 4.0b2

Cleaned up router/parser interactions. Jess will now read only one construct on a line of input (just like CLIPS). All Jess output now goes through WSTDOUT router, not through ReteDisplay.stdout(). Fixed bug whereby second and later references to subfields of multifields on the LHS of a rule would resolve to the whole multifield. modify can now properly handle multislots. format handles trailing spaces. Finally, parsing of integers: 2 is an RU.INTEGER, while 2.0 is an RU.FLOAT. Added eval and list-function$.

Version 4.0b1

Code reformat. Major performance enhancements (Value and Funcall recycling; Fastfunction interface; Rete memories are now btrees; RU.CLEAR tokens). test CE. Return-value constraints. ppdefrule thanks to Rajaram Ganeshan. Blank variables in not CEs. system blocks by default. readline fixed. build supported. logic for predicate functions in Rete network now precisely the same as for CLIPS. QuizDisplay demo. while and if accept boolean variables. Implied returns from if and while functions. Added explode$. Added I/O routers: open, close. Added format. Added bag.

Version 3.2

system and integer Userfunction classes renamed (Win95 filename capitalization problem!). Broken delete$, insert$, replace$ fixed. view command added. Big if/then in Funcall class finally removed in favor of separate implementation classes for intrinsics, leading to a modest speed increase. Documentation vastly expanded! Added catch for ArrayOutOfBoundsException in command-line interface; no more crash on wrong number of arguments. Broken evenp, oddp fixed. str-cat, sym-cat made more general. Broken sub-string fixed. Big switch in Node1 class replaced by separate classes, leading to a very modest speed increase.

Version 3.1

Added the assert-string and batch commands. Two bug fixes in multislot code (thanks to Nancy Flaherty). Added undefrule and the ability to redefine rules. Added the system function, although it doesn't work very well under Java. Public function engine() in jess.Context class allows you to do fancier things in Userfunctions. Added the non-standard load-package and load-function functions. Many new contributed functions packaged with Jess for doing math, handling multifields, and other neat stuff thanks to Win Carus. Added time (1 second resolution).

Version 3.0

A few code changes to accomodate Microsoft's Java compiler; Jess now compiles unchanged with JVC thanks to Mike Finnegan. Added member$ multifield function. Added clear intrinsic thanks to Karl Mueller. Introduced a new way of handling not patterns which I think finally guarantees there are no more not-related bugs remaining! load-facts, which has been non-functional throughout the beta period, is working again. Documentation now explains unzipping and compiling a little better. Modified the way fact-id's are handled so that you can write (retract 3) to retract fact #3.

Version 3.0b2

Lots of bug reports and improvement suggestions from the field - thanks folks! All the reported bugs in the multifield implementation, and some residual odd behavior in the not CE, have been fixed. The exit command has been added. A command prompt has been added. The # character can now be used in symbols. The access levels on some methods in the Rete class have been opened up; Rete is no longer final. nth$ is now 1-based, as it is in CLIPS. The if and while constructs now fire on not FALSE instead of TRUE. The str-index function has been fixed and added. Probably a few more things I'm forgetting here. Thanks for the input. Particular thanks to Nancy Flaherty, Joszef Toth, Karl Mueller, Duane Steward, and Michelle Dunn for reporting bugs fixed in this version; sorry if I left anyone out.

Version 3.0b1

First public release of Jess 3.0.

Version 3.0a3

UserPackage interface. Lots of new example UserFunctions for multifields, string, and predicates.

Version 3.0a2

Multislots! Also important bug fix: under certain circumstances, the Rete network compilation could fail 1) if (not()) CEs occurred on the LHS of a rule, 2) new variables were introduced in that rule's patterns listed after the (not()) CEs, and 3) these latter variables were tested (i.e., in a predicate constraint) on the LHS of the rule.

Version 3.0a1

Incremental reset. Watch activations. gc() in LostDisplay, NullDisplay. Multifields! All the Rete engine classes are now in a package named jess. Many classes and methods that should not be manipulated by clients are now package-private.

Version 2.2.1

Ken Bertapelle found another bug, which has been squashed, in the pattern network.

Version 2.2

Jess 2.2 adds a few new function calls (load-facts, save-facts) and fixes a serious bug (thanks to Ken Bertapelle for pointing it out!) which caused Jess to crash when predicate constraints were used in a certain way. Another bug fix corrected the fact that retract only retracted the first of a list of facts. Jess used to give a truly inscrutable error message if a variable was first used in a not CE (a syntax error); the current error message is much easier to understand. I also clarified a few points in the documentation.

Version 2.1

Jess 2.1 is *much* faster than version 2.0. The Monkey example runs in about half the time as under Jess 2.0, and for some inputs, the speed has increased by an order of magnitude! This is probably the last big speed increase I'll get. For Java/Rete weenies, this speed increase came from banishing the use of java.lang.Vector in Tokens and in two-input node memories. Jess is now within a believable interpreted Java/C++ speed ratio range of about 30:1. Jess 2.1 now includes rule salience. It also implements a few additional intrinsic functions: gensym*, mod, readline. Jess 2.1 fixes a bug in the way predicate constraints were parsed under some conditions by Jess 2.0. The parser now reports line numbers when it encounters an error.

Version 2.0

Jess 2.0 is intrinsically about 30% faster than version 1.0. The internal data structures changed quite a bit. The Rete network now shares nodes in the Join network instead of just in the pattern network. The current data structures should allow for continued improvement.

Back to index

12. References

Eventually this should be an extensive list of useful references.

12.1. Java and Java Programming

- Java's home page

12.2. Expert Systems

- Giarratano and Riley, "Expert Systems: Principles and Programming", Second Edition; ISBN 0878353356.
- The CLIPS home page.
- Stuart Russell and Peter Norvig, "Artificial Intelligence - A Modern Approach," ISBN 0131038052.
- John Durkin, "Expert Systems - Design and Development," ISBN 0023309709.
- Guus Schreiber, Hans Akkermans, Anjo Anjewierden, Robert de Hoog, Nigel Shadbolt, Walter Van de Velde and Bob Wielinga, "Knowledge Engineering and Management - The Common KADS Methodology," ISBN 0262193000.
- Nils J. Nilsson, "Principles of Artificial Intelligence," ISBN 0934613109.
- Jay Aronson and Efraim Turban, "Decision Support and Intelligent Systems", ISBN 0137409370.
- Mark Watson, "Intelligent Java Applications for the Internet and Intranets"

Back to index

13. Release Notes

13.1. Important changes in Jess 6.1

These are a few of the things that changed between Jess 6.0 and 6.1 that are of particular interest. Also see the change log for more information.

- The `unique` CE has been removed. For the time being, the parser will accept but ignore it. If your program depended on `unique` for correct operation, it may need to be rewritten. Note that the manual has always discouraged use of `unique` for anything other than optimization.
- The `assert` method in the `Rete` class is deprecated; use `Rete.assertFact()` instead. `assert` will be removed in the first post-6.1 release.

13.2. Porting from Jess 5 to Jess 6

Jess 6 introduces many new features, but it is mostly backwards compatible with Jess 5. This section lists a few specific areas where Jess 5 applications may need to be changed to work with Jess 6.

- **Java 2 is required.** Jess 6 now requires a Java 2 compatible JVM (JDK 1.2 or later.) Jess 5 worked with JDK 1.1.
- **Batch files in JAR files.** The rules for loading a batch file from a JAR have changed. See the javadoc for java.lang.Class.getResource() for the appropriate way to reference a batch file in a JAR.
- **Strategy interface changed.** If you wrote any custom conflict resolution strategies for Jess 5, they need to be rewritten from scratch for Jess 6.
- **No functions are optional.** Under Jess 5, you had to load a number of optional Userpackages using Rete.loadPackage(). In Jess 6, there are -no- optional functions; everything is loaded by default. Loading packages redundantly does no harm, it's just inefficient. More seriously, some of the old optional packages either no longer exist, or no longer support the Userpackage interface. Simply delete any code you have in your app which calls Rete.addUserpackage() on a Jess-supplied package. In Jess 6.1, these classes are no longer public.
- **Script library loaded automatically.** Under Jess 5, you had to load the Jess script library manually; in Jess 6, it happens automatically. There's no harm in loading it redundantly, but there's no value in it, either.
- **Pattern bindings are `jess.Fact` objects.** Perhaps the biggest change is that pattern bindings, like `?var` in

  ```
  ?var <- (pattern)
  ```

  are `jess.Fact` objects in Jess 6, not just integers as they were in Jess 5. That means that Jess 5 code like

  ```
  ;; Get the 'head' of a fact
  (get (call (engine) findFactById ?var) name)
  ```

  won't work. Instead, it can be replaced by the much simpler and much more efficient

```
;; Get the 'head' of a fact
(get ?var name)
```

Some fairly widely distributed code needs to be updated in this way.

13.3. Past problems now fixed

The issues in this list were dealt with during the 6.1 development cycle and aren't problems any more.

- **`breadth` conflict resolution strategy.** Both built-in conflict-resolution strategies are now correct under all circumstances. Recall that the order of firing of rules activated by the same fact, with the same salience, is arbitrary. You may now also notice slight differences in this arbitrary ordering compared to previous versions of Jess.
- **`logical` facility Java interface** There is now a public Java API to the logical dependency facility. There are two Jess functions, `dependencies` and `dependents`, that give you logical information about a specific fact.
- **Backwards chaining during `run-query`** You can now control how many rules can be fired during The `run-query` using the max-background-rules declaration.
- **Performance issues** The performance issues with earlier 6.1 alpha versions of Jess have been fixed. Jess 6.1 is faster than Jess 6.0 on many problems.
- **Classloader issues** The Jess library now works perfectly if you install jess.jar as a standard extension under Java 2. You must use the "app object" constructor for the Rete class if you want Jess to be able to use reflection to manipulate classes from your class path. The classes `jess.Main` and `jess.Console` will work if you've installed them this way, but they *won't* be able to see any classes loaded via your classpath. This is an ongoing issue.

  The "app object" constructor now tells Jess only where to load user classes; it always loads its own scripts and classes from the class loader that loads `jess.Rete`.

  The manual doesn't yet mention any of this class loader stuff.
- **Defclass and Deftemplate inheritance** `defclass` and `deftemplate` constructs can inherit from each other to the extent that this makes sense.

Back to index

Appendix A. Jess Functions Grouped by Usage

Miscellaneous functions

bag, bind, clear-storage, fetch, gensym*, jess-version-number, jess-version-string, set-factory, setgen, store, system, time

Functions related to files and other I/O

close, format, get-multithreaded-io, open, printout, read, readline, set-multithreaded-io, socket

Functions for interacting with Java code

call, context, defclass, definstance, get, get-member, import, instanceof, load-function, load-package, new, set, set-member, synchronized, throw, try, undefinstance, update

Logical functions

and, bit-and, bit-not, bit-or, not, or

Functions used primarily for debugging

list-function$, matches, show-jess-listeners, unwatch, view, watch

Functions for working with defqueries

count-query-results, run-query

Control structures

apply, call-on-engine, foreach, if, progn, return, while

Functions for pretty-printing constructs

list-deftemplates, ppdeffacts, ppdeffunction, ppdefglobal, ppdefquery, ppdefrule, ppdeftemplate, rules, show-deffacts, show-deftemplates

Functions for interacting with the rule engine

agenda, batch, bload, bsave, build, clear, clear-focus-stack, defadvice, do-backward-chaining, engine, eval, exit, focus, get-current-module, get-focus, get-focus-stack, get-reset-globals, get-salience-evaluation, get-strategy, halt, list-focus-stack, pop-focus, reset, run, run-until-halt, set-current-module, set-node-index-hash, set-reset-globals, set-salience-evaluation, set-strategy, undefadvice, undefrule

Mathematical functions

*, **, +, -, /, <, <=, <>, =, >, >=, abs, div, e, eq, eq*, exp, float, integer, log, log10, long, max, min, mod, neq, pi, random, round, sqrt

Functions which query the type of a Value

evenp, external-addressp, floatp, integerp, lexemep, longp, multifieldp, numberp, oddp, stringp, subsetp, symbolp

String manipulation functions

asc, lowcase, str-cat, str-compare, str-index, str-length, sub-string, sym-cat, upcase

Functions for working with multifields

complement$, create$, delete$, explode$, first$, implode$, insert$, intersection$, length$, member$, nth$, replace$, rest$, subseq$, union$

Functions for working with facts

assert, assert-string, dependencies, dependents, duplicate, fact-id, fact-slot-value, facts, load-facts, modify, retract, retract-string, save-facts

Back to index